Advances in Type Systems for Functional Logic Programming

Proyecto Fin de Máster en Programación y Tecnología Software

Máster en Investigación Informática, Facultad de Informática, Universidad Complutense de Madrid

Author: **Enrique Martín Martín**
Director: **Dr. Francisco Javier López Fraguas**
2008-2009

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Advances in Type Systems for Functional Logic Programming", realizado durante el curso académico 2008-2009 bajo la dirección de D. Francisco Javier López Fraguas en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Fdo: Enrique Martín Martín

Summary

Declarative languages provide a higher and more abstract level of programming than traditional imperative languages. Functional and logic languages are the two most important declarative programming paradigms, and their combination has been a topic of research in the last two decades. Functional logic languages have inherited the classical Damas & Milner type system [20] from their functional part, due to its simplicity and popularity, but this naive approach does not work properly in all cases. It is known that higher order patterns (HO) in left-hand sides of program rules may produce undesirable effects from the point of view of types. In particular some expressions can lose their type after a step of computation. In this work we propose an extension to the classical Damas & Milner type system that fixes this situation. This problem was first detected in [24], and the proposed solution was forbidding opaque HO patterns. We propose a more relaxed way of tackle this problem, making a distinction between *transparent* and *opaque* variables. A variable is transparent in a pattern if its type is univocally fixed by the type of the pattern, and opaque otherwise. We prohibit only the occurrence of opaque variables when they are used in the right-hand side of the rules. We have developed the type system trying to clarify the behavior (from the point of view of types) that local definitions have in different implementations of functional and functional logic languages. This is an issue that varies greatly, and it is not usually well documented or formalized. Apart from the type system we also present type inference algorithms for expressions and programs, and provide a prototype of implementation in Prolog that will be soon integrated into the $\mathcal{TOY}$ [45, 69] compiler. We have paid special attention to the formal aspects of the work. Therefore we have developed detailed proofs of the properties of the type system, in particular the *subject reduction* property (expressions keep their type after evaluation), and the soundness and completeness of the inference algorithms wrt. the type system.

Keywords

Type systems, functional logic programming, higher order patterns, opaque patterns, parametric polymorphism, local definitions, $\mathcal{TOY}$.

Resumen

Los lenguajes declarativos proporcionan un nivel de programación más alto y abstracto que los lenguajes imperativos tradicionales. Los lenguajes funcionales y los lógicos son los dos paradigmas declarativos más importantes, y su combinación ha sido un tema de investigación en las últimas dos décadas. Los lenguajes lógico funcionales han heredado el clásico sistema Damas & Milner [20] de su parte funcional, debido a su simplicidad y popularidad, pero esta aproximación directa no funciona correctamente en todos los casos. Es conocido que los patrones de orden superior en los lados izquierdos de las reglas de programa pueden producir efectos no deseados desde el punto de vista de los tipos. En particular algunas expresiones pueden perder su tipo tras un paso de cómputo. En este trabajo proponemos una extensión del clásico sistema de tipos Damas & Milner que arregla esta situación. Este problema fue detectado por primera vez en [24], y la solución propuesta fue prohibir los patrones de orden superior opacos. Nosotros proponemos una forma más relajada de abordar este problema, haciendo una distinción entre variables *transparentes* y *opacas*. Una variable es transparente en un patrón si su tipo está unívocamente determinado por el tipo del patrón, y es opaca en otro caso. Nosotros prohibimos solamente la aparición de variables opacas cuando son usadas en los lados derechos de las reglas. Hemos desarrollado el sistema de tipos tratando de clarificar el comportamiento (desde el punto de vista de los tipos) que las declaraciones locales tienen en diferentes implementaciones de lenguajes funcionales y lógico funcionales. Éste es un aspecto que varía mucho, y usualmente no está bien documentado ni formalizado. Aparte del sistema de tipos presentamos algoritmos de inferencia de tipos para expresiones y programas, y proporcionamos un prototipo de implementación en Prolog que será integrado próximamente en el compilador de $\mathcal{TOY}$ [45, 69]. Hemos prestado especial atención a los aspectos formales del trabajo. Por ello hemos desarrollado demostraciones detalladas de las propiedades del sistema de tipos, en particular de la propiedad de *preservación del tipo* o *subject reduction* (las expresiones mantienen su tipo tras la evaluación), y la corrección y completitud de los algoritmos de inferencia con respecto al sistema de tipos.

Palabras clave

Sistemas de tipos, programación lógico funcional, patrones de orden superior, patrones opacos, polimorfismo paramétrico, definiciones locales, $\mathcal{TOY}$.

Agradecimientos

Gracias a Paco por muchas cosas. Primero por haberme enseñado algunas de las asignaturas más interesantes de la carrera, y por ofrecerme la oportunidad de quedarme en la facultad tras terminar la carrera (e insistir en ello). Si no fuera por él seguramente ahora mismo no me dedicaría a la investigación, sino a alguna labor aburrida y estresante en la empresa. Gracias también a él y Juan por haberme ido introduciendo poco a poco en el mundillo de la investigación en estos dos años, y por todas las reuniones que hemos tenido (y las que presumiblemente nos quedan). También agradecer al "núcleo duro" del despacho 220 por crear un ambiente tan bueno, que hace que venir a trabajar sea muchas veces un placer, especialmente al "*Rincón de Nacho*"[1], oráculo, centro de reuniones y peaje obligatorio para todo habitante del 220. Y gracias a Sonseca (Toledo) y a toda la gente que allí tengo porque dos días a la semana puedo olvidarme de todo lo que hago en los otros cinco :-).

Pero sobre todo gracias a Ester y a mi familia, porque sin ellos nada de todo esto que hago tendría mucho sentido.

[1] situado en el fondo Norte

Contents

List of Figures

$X, Y, Z \ldots$	Data variable	Figure 2.1
c	Data constructor	Figure 2.1
f	Function	Figure 2.1
h	Data constructor or function symbol	Figure 2.1
s	Data variable, data constructor or function	Figure 2.1
t	Pattern	Figure 2.1
e	Expression	Figure 2.1
$\alpha, \beta, \gamma \ldots$	Type variable	Figure 2.3
τ	Simple type	Figure 2.3
σ	Type-scheme	Figure 2.3
$\mathcal{A}$	Set of assumptions	Definition 8
θ	Data substitution	Section 2.1
π	Type substitution	Section 2.2
$\vdash$	Typing relation	Figure 3.1
$\vdash^{\bullet}$	Extended typing relation	Figure 3.2
$\Vdash$	Type inference relation	Figure 4.1
$\Vdash^{\bullet}$	Extended type inference relation	Figure 4.2
$\mathcal{E}$	Type inference algorithm	Figure 4.3
$\mathcal{E}^{\bullet}$	Extended type inference algorithm	Figure 4.4
$\Pi^{e}_{\mathcal{A}}$	Typing substitutions wrt. $\vdash$	Definition 16
${}^{\bullet}\Pi^{e}_{\mathcal{A}}$	Typing substitutions wrt. $\vdash^{\bullet}$	Definition 16
$\succ$	Generic instance relation	Definition 5
$Gen(\tau, \mathcal{A})$	Generalization of a type	Definition 11
$\succ_{var}$	Variant relation	Definition 7
$\mathcal{P}$	Program	Section 2.1
$wt_{\mathcal{A}}(\mathcal{P})$	Well-typed program wrt. $\mathcal{A}$	Definition 15

Figure 1: **Notation**

Chapter 1

Introduction

In this work we will contribute two advances on types systems for functional logic languages that we have developed. The main advance was developed to avoid an undesirable behavior of actual type systems for functional logic languages. These languages have inherit the classical Damas & Milner type system, but it does not work correctly in the presence of *HO patterns*, i.e., patterns built up as partial application of function or constructor symbols. As we will explain later, with the classical type system some expressions can lose their type after a step of computation. The second advance was developed trying to clarify the behavior (from the point of view of types) that local definitions have in different implementations of functional and functional logic languages. This is an issue that varies greatly, and it is not usually well documented or formalized. In this work we propose an extension to the classical Damas & Milner type system [20] that tackles these two problems. We also present type inference algorithms for expressions and programs, and provide a prototype of implementation in Prolog that will be soon integrated into the $\mathcal{TOY}$ [45, 69] compiler. We have paid special attention to the formal aspects of the work. Therefore we have developed detailed proofs of the properties of the type system, the *subject reduction* property (expressions keep their type after evaluation), and the soundness and completeness of the inference algorithms wrt. the type system.

The results of this work have been accepted as a paper in the 18^{th} International Workshop on Functional and (Constraint) Logic Programming (WFLP'09) held in Brasilia on June 28th. The paper has been also accepted to appear in the Lecture Notes in Computer Science volume associated to the workshop.

This introduction chapter offers a global vision of type systems (Section 1.1) and functional logic languages (Section 1.2), a motivation of the work (Section 1.3) and a summary of the main contributions (Section 1.4). The rest of the work is organized as follows. Chapter 2 contains some preliminaries

about functional logic languages and types, including the notation used. In chapter 3 we expose the type system and prove its soundness wrt. the *let*-rewriting semantics of [44]. Chapter 4 contains a type inference relation and algorithm that let us find the most general type of expressions. Chapter 5 presents a method to infer types for programs, with two different flavors. These algorithms for inferring types for expressions and programs have been implemented as a prototype of type system, ready to be integrated into $\mathcal{TOY}$. Chapter 6 explains in detail the most important parts of this implementation. Finally, Chapter 7 contains some conclusions, contributions and future work. Complete proofs of the theoretical results of this work have been included in Appendix A.

1.1 Type systems

In general, type systems can be viewed as a family of program analysis, like abstract interpretation or data-flow analysis. A suitable definition of this family can be found in [58]:

> "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

As a simple example, a type system can consider expressions like $true$, $not\ X$ or $Y \wedge false$ to have the same type: $boolean$; and expressions like 0, $X * X$ or $0 + 1$ to have type $integer$. Then it may reject programs where it expects an integer expression and finds a boolean expression. This could happen in $(0 + 1) \wedge true$, where the type system expects both arguments of $\wedge$ to have a boolean type, but it finds the first argument $0 + 1$ has type integer.

A type system provides some benefits to a programming language. The most important are:

- **Safety**. Type systems usually assure the absence of some run-time errors, detecting and rejecting programs which contain parts that may create problems during execution. Examples of these problematic parts can be "meaningless" expressions like `(0+1) ∧ true` or `3 / "hello"`.
- **Code clarity**. A type system forces programmers to write code in a certain way, providing homogeneity and clarity to the pieces of code. Types can be part of the documentation of a program as well. In particular explicit type declarations in functions provide valuable information about the meaning of a function without the need of looking at the code. These declarations have the feature that they are never out of date, as usually happens with comments embedded in functions, because the type system checks them in every compilation.

- **Efficiency**. Efficiency can be other of the provided benefits. A type system recollects information about the program, and this information can be valuable for optimizations. One example is the type system of Fortran, which was introduced to improve the efficiency of numerical calculations by distinguishing between integer and real arithmetic expressions. Another example are region inference algorithms, which can reduce the garbage collection during execution [71].

Type systems can be classified depending on which moment the type checking is performed. A *static type system* is that which performs all the type checking in compile time. On the other hand, a *dynamic* type system may (or may not) perform some type checking compile time, but also needs run-time checks. Static type systems increases the compile time of programs, but unlike dynamic type systems they do not penalize the execution of programs with extra checks. However, dynamic type systems can delay to run-time checks some difficult (or even undecidable) checks, accepting programs that a static type system must reject.

A vast survey about type systems, their history and their formalization can be found in [13] and [14].

1.1.1 Overview of type systems

In this section we will try to give an overview of type systems. We will pay special attention to Damas & Milner type system, the most famous type system in functional and functional logical languages, and the one we have extended in this work.

Brief history about type systems

Types and type systems have been present in programming languages since the very beginning. The first appearance was in 1954 in Fortran. As we have explained before, types were introduced to distinguish between integer and floating-point numbers, to take advantage of the different specialized hardware. Fortran accomplished this distinction by the first letter of variable names. Algol60 was the first language to have an explicit notion of type and the associated conditions for type checking. It supported integers, reals and booleans. In the 1960s, this notion of type was extended to richer classes of constructions. Pascal extended the notion of type to arrays, record and pointers, and also supported user-defined types. However, it did not define the equivalence between types and left some ambiguities and insecurities. Algol68 had a more rigorous notion of type than Pascal, with a relation of equivalence and a rich variety of types. It had a decidable type checking algorithm, but it was so complex and difficult to understand that it was considered to be a flaw, resulting in a reaction against complex type systems. Simula was the

first *object-oriented* language. Its types included classes and subclasses, but it did not have the notion of information hiding, which was introduced in subsequent object-oriented languages like Smalltalk or Loops. Modula-2 was the first to use modularization as a structuring principle. Then, types could be made opaque interfaces to achieve data abstraction. ML [31], the language for proof tactics in the Edinburgh LCF theorem prover, introduced the notion of *parametric polymorphism*, as well as a type system with principal types and a simple type checking algorithm. This algorithm had the ability of inferring types automatically when no explicit type declaration was provided. The simplicity of this type system (the famous Damas & Milner type system) and the automatic inference of types was the key of its success, and it is still present in current functional languages like Haskell [54], Clean [11, 60] or F# [2]. It has also been inherited as the type system for functional logic languages like $\mathcal{TOY}$ [45, 69] or Curry [30].

In spite of the benefits provided by type systems, some languages have decided not to have type checking at all. This is the case of Erlang [1, 15], a functional language developed by Ericsson Computer Science Laboratory and used to build concurrent and distributed systems. Prolog [21, 66] is also a language that historically has not supported types. Implementations like SICStus [5] or SWI Prolog [73, 6] follow this philosophy, although Ciao Prolog [12] supports types as assertions that are checked by the system (static or dynamically).

Type systems for programming languages, specially for functional languages, are an active area of research [58, 59]. Examples of these developments are type classes [72, 52, 51], arbitrary-rank polymorphism [56], Generalized Algebraic Data Types (GADTs) [16, 61, 57] or dependent types [47].

The Damas & Milner type system

The Damas & Milner type system is one of the most famous type systems in functional programming languages and it has been inherited in functional logic languages. It was first developed by Robin Milner in 1978 [48]. In that paper the author presented a type system for polymorphic procedures as well as a compile time type-checking algorithm $\mathscr{W}$. He also showed that well-typed programs cannot "go wrong", and proved the soundness of $\mathscr{W}$. As the author mentioned in the introduction, after doing this work he became aware of Hindley's [33] method for deriving the "principal type-scheme" for terms in combinatory logic. Hindley had been the first to notice that the Unification Algorithm [63] was appropriate for this problem. However, the work of Milner can be regarded as an extension of Hindley's method to programming languages with local definitions, and as semantic justification of the method. In 1982, with the help of Luis Damas –a Milner's PhD student–, they proved that $\mathscr{W}$ was also complete [20, 19], i.e., that the type checking algorithm found the most general type possible for every expression.

Due to its origins, this type system is called Hindley-Milner, or Damas-Milner, or even Hindley-Milner-Damas. In this work we will call this type system Damas & Milner, since their implementation of the algorithm $\mathscr{W}$ in ML made the type system popular.

The Damas & Milner type system has a series of interesting features which are the cause of its success:

- **Simplicity**. The type system is compound by six rules. This makes it is easy to understand and predict the expected types of the expressions.
- **Parametric polymorphism [67]**. An expression can have multiple types, but all of them are instances of a parametric one. An example is the empty list constructor [ ]. This expression can have types $[bool]$, $[int]$, $[[int]]$... but they are instances of $[\alpha]$, where α is the parameter. The same happens with the well-known function map, which has type $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$. It does not support *ad-hoc polymorphism*, where expressions can have different types but they are not related by this uniform parametricity. An example of this other kind of polymorphism is $(+)$, which can have types $int \rightarrow int \rightarrow int$ or $real \rightarrow real \rightarrow real$ but it cannot have a type $[bool] \rightarrow [bool] \rightarrow [bool]$.
- **Principal types**. Every expression has a most general type. This type represents all the other possible types.
- **Syntactic soundness**. It states that well-typed programs cannot "go wrong", i.e., if a program is considered as well-typed, its execution will not yield certain undesired situations.
- **Inference algorithm**. There exists a simple algorithm ($\mathscr{W}$) to find the principal type of an expression. It is stronger than a simple type checker, since it also finds the type when it is not explicitly given. Therefore it permits the programmer to omit much type declarations. The algorithm is sound and complete, and does all the work in compile time.

1.2 Functional logic programming

Declarative languages provide a higher and more abstract level of programming than traditional imperative languages. In contrast with them, declarative languages describe *what* are the properties of the problem and the expected solutions, instead of *how* to obtain them. Functional and logic languages are the two most important declarative programming paradigms. Functional languages are based on λ-calculus and term rewriting, and consist on functions defined by equations that are used from left to right.

These languages provide useful concepts to the programmer, like generic programming (using higher-order functions and polymorphic types). Logic languages are based on (a subset of) predicate logic, and their execution model is goal solving based on resolution. They also provide useful concepts like computation with partial information (logic variables) and nondeterministic search for solutions. Therefore their combination is an interesting goal, and has been a topic of research in the last two decades.

The combination of them can be tackled in different ways. Logic languages can be extended with features for functional programming by means of syntactic sugar to support functional notation, which is translated by some preprocessor. This is the case of Ciao-Prolog [12]. Mercury [65] can also be included in this family. On the other hand, logic programming aspects can be integrated into functional programming combining the resolution principle with some sort of function evaluation, trying to retain the efficient demand-driven computation strategy of purely functional computations. Into this family we can remark Escher [43], $\mathcal{TOY}$ [45, 69] or Curry [30]. In Escher, function calls are suspended if they are not instantiated enough for deterministic reduction. $\mathcal{TOY}$ and Curry overcome this limitation treating functions like predicates, with unknown arguments that are instantiated to be able to apply a rule. This method, known as *narrowing*, combines the functional concept of function reduction with the logical ones of unification and nondeterministic search.

The expressiveness of functional logic languages is illustrated in the following examples, written using $\mathcal{TOY}$ syntax. The first is a pretty inefficient method for sorting lists: *permutsort*. With this method permutations of the original list are calculated, and the first that is sorted is returned.

Example 1 (Permutation sort)**.**

```
insert X Ys     = [X|Ys]
insert X [Y|Ys] = [Y|insert X Ys]

permute []     = []
permute [X|Xs] = insert X (permute Xs)

leq zero Y            = true
leq (succ X) zero     = false
leq (succ X) (succ Y) = leq X Y

sorted []        = true
sorted [X]       = true
sorted [X,X2|Xs] = true <== leq X X2 == true, sorted [X2|Xs]
```

```
check L = L <== sorted L == true
sort L = check (permute L)
```

`insert` is a function that inserts an element nondeterministically in a list, so `permute` is also nondeterministic. `sorted` is a deterministic function that checks if the given list is sorted, and `check` is a function that returns the list passed as an argument if it is sorted. Finally, `sort` calculates a permutation of `L`, and returns it if it is sorted. This example shows how the combination of nondeterminism (from logic programming) and lazy evaluation (from functional programming) improves the efficiency. In a purely logic language like Prolog, the same example is written as:

```
permutationSort(L,L2) :- permute(L,L2), sorted(L2).
```

Here, every candidate solution `L2` is completely created before it is tested. In a purely functional language, one usually follows the 'list of successes' approach: generate the list of all possible solutions (the permutations) and filter it (only those sorted). In the functional logic setting we still use a generator to create the possible solutions one by one, as in a logic language. But the laziness allows us to generate only a small fraction of the list, the minimum to be able to apply `sorted`. This way, the creation of a list will be interrupted as soon as `sorted` recognizes it cannot lead to an ordered list. Therefore we do not need to generate the whole list of candidates, as in a functional language, and we avoid checking different list with the same prefix; obtaining an important improvement in efficiency. For more information about this example, see [25].

The following is another example of the expressiveness of functional logic languages. We can specify the behavior of a function `last` that returns the last element of a list as: $last\, l = e$ iff $\exists xs.\, append\, xs\, [e] = l$. Using a functional logic language we can easily define a function `last`:

Example 2 (Last element of a list [29])**.**

```
append [] Ys = Ys
append (X:Xs) Ys = X : (append Xs Ys)

last L = E <== (append Xs [E]) == L
```

Here we define `append`, the concatenations of lists, in the usual way. The function `last` is defined directly from the specification, by means of a conditional equation. This equation states that the last element of a list `L` is `E` if `append Xs [E]` is equal to `L`, where `Xs` and `E` are extra variables, i.e.,

variables that appear in the right-hand side of the definition and not in the left-hand side. In this case, functional logic languages provide search for solutions for these extra variables. For example, if we want to reduce `last [1,2,3]` the system will instantiate `Xs` to `[1,2]` and `E` to `3`, which satisfies the condition $append\ [1,2]\ [3] = [1,2,3]$; returning `3` as the last element.

Two surveys covering the integration of functional and logic languages can be found [28] and [29]. Besides, more information about functional logic programming in general can be found in Michael Hanus' web page [27].

1.3 Motivation

In this section we will explain the motivation of this work. As we have said before, the main motivation is to solve a known problem with higher order (HO) patterns in functional logic languages. In the previous work "Polymorphic Types in Functional Logic Programming" [24] this problem was detected, and the solution was to forbid potentially problematic HO patterns. In this work we propose a more flexible solution that admits patterns which were forbidden in [24]. A secondary motivation is to technically clarify the different grade of polymorphism that can be given to local declarations. Implementations of functional and functional logic languages vary significantly in this point, and they do not usually explain nor formalize their choice.

1.3.1 Type problems with HO patterns in FLP

In our formalism patterns appear in the left-hand side of rules and in lambda or let expressions. Some of these patterns can be HO patterns, if they contain partial applications of function or constructor symbols. HO patterns can be a source of problems from the point of view of the types. In particular, it was shown in [24] that unrestricted use of HO patterns leads to loss of *subject reduction*, an essential property for a type system expressing that evaluation does not change types. The following is a crisp example of the problem.

Example 3 (Polymorphic Casting [10])**.**
Consider the program consisting of the rules $\{snd\ X\ Y \to Y, and\ true\ X \to X, and\ false\ X \to false\}$ *with the usual types inferred by a classical Damas & Milner algorithm:* $\{snd : \forall\alpha,\beta.\alpha \to \beta \to \beta, and : bool \to bool \to bool\}$*. Then we can write the functions* $co\ (snd\ X) \to X$ *and* $cast\ X \to co\ (snd\ X)$*, whose inferred types will be* $\forall\alpha.\forall\beta.(\alpha \to \alpha) \to \beta$ *and* $\forall\alpha.\forall\beta.\alpha \to \beta$ *respectively. It is clear that the expression* $and\ (cast\ 0)\ true$ *is well-typed, because* $cast\ 0$ *has type* $bool$

(in fact it has any type), but if we reduce that expression using the rule of $cast$ *the resulting expression* $and\ 0\ true$ *is ill-typed.*

In this case $cast$ *behaves as the usual identity function, returning the same element that accepts as argument. But it permits us to "cast" the type of the element to* ***any other*** *type, making the type system pretty useless.*

The problem arises when dealing with HO patterns, because unlike FO patterns, knowing the type of a HO pattern does not always permit us to know the type of its subpatterns. For example, given the FO pattern $(X, true)$ of type $(int, bool)$, we know that the type of X must be int. The same happens with $[X, 0]$ of type $[int]$. On the other hand, knowing that the HO pattern $snd\ X$ has type $int \to int$ does not permit us to know anything about the type of X. X may have type int, $bool$, $[int \to bool]$ or any other type, and the type of the whole patter will not change: $int \to int$. In the previous example the cause of the type problems is function co, because its pattern $snd\ X$ is *opaque* and shadows the type of its subpattern X. Usual inference algorithms treat this opacity as polymorphism, and that is the reason why it is inferred a completely polymorphic type for the result of the function co.

In [24] the appearance of any opaque pattern in the left-hand side of the rules is prohibited, but we will see that it is possible to be less restrictive. The key is making a distinction between **transparent** and **opaque** variables of a pattern: a variable is transparent if its type is univocally fixed by the type of the pattern, and is opaque otherwise. We call a variable of a pattern **critical** if it is opaque in the pattern and also appears elsewhere in the expression. The formal definition of opaque and critical variables will be given in Chapter 3. With these notions we can relax the situation in [24], prohibiting only those patterns having critical variables.

1.3.2 Variety of polymorphism in local definitions

Functional and functional logic languages provide syntax to introduce local definitions inside an expression. But in spite of the popularity of let expressions, different implementations treat them differently because of the polymorphism they give to bound variables. This difference can be observed in Example 4, being $(e_1, \ldots, e_n)$ and $[e_1, \ldots, e_n]$ the usual tuple and list notation respectively.

Example 4 (let expressions)**.**

$$e_1 \equiv let\ F = id\ in\ (F\ true, F\ 0)$$
$$e_2 \equiv let\ [F, G] = [id, id]\ in\ (F\ true, F\ 0, G\ 0, G\ false)$$

Intuitively, e_1 gives a new name to the identity function and uses it twice with arguments of different

types. Surprisingly, not all implementations consider this expression as well-typed, and the reason is that F is used with different types in each appearance: $bool \to bool$ and $int \to int$. Some implementations as Clean 2.2, PAKCS 1.9.1 or KICS 0.81893 consider that a variable bound by a let expression must be used with the same type in all the appearances in the body of the expression. In this situation we say that lets are completely monomorphic, and write let_m for it.

On the other hand, we can consider that all the variables bound by the let expression may have different but coherent types, i.e., are treated polymorphically. Then expressions like e_1 or e_2 would be well-typed. This is the decision adopted by Hugs Sept. 2006, OCaml 3.10.2 or F# Sept. 2008. In this case, we will say that lets are completely polymorphic, and write let_p.

Finally, we can treat the bound variables monomorphically or polymorphically depending on the form of the pattern. If the pattern is a variable, the let treats it polymorphically, but if it is compound the let treats all the variables monomorphically. This is the case of GHC 6.8.2, SML of New Jersey v110.67 or Curry Münster 0.9.11. In this implementations e_1 is well-typed, while e_2 not. We call this kind of let expression let_{pm}.

Programming language and version	let_m	let_{pm}	let_p
GHC 6.8.2		×	
Hugs Sept. 2006			×
Standard ML of New Jersey 110.67		×	
Ocaml 3.10.2			×
F# Sept. 2008			×
Clean 2.0	×		
$\mathcal{TOY}$ **2.3.1***	×		
Curry PAKCS 1.9.1	×		
Curry Münster 0.9.11		×	
KICS 0.81893	×		

(*) we use `where` instead of `let`, not supported by $\mathcal{TOY}$

Figure 1.1: **Let expressions in different programming languages**

Figure 1.1 summarizes the decisions of various implementations of functional and functional logic languages. The exact behavior wrt. types of local definitions is usually not well documented, not to say formalized, in those systems. A sample of this can be found in GHC [3], a famous implementantion of Haskell. They treated pattern bindings polymorphically (let_p) but in July 2006 Simon Peyton Jones changed experimentally the behavior to let_{pm}, in order to check if people noticed it (and he only received a few messages complaining). They have finally adopted the new choice, as can be seen in [55, 4]. Our aim is to technically clarify this question by adopting a neutral position, and formalizing the different possibilities for the polymorphism of local definitions.

1.4 Contributions

In this work we have proposed a type system for functional logic languages based on Damas & Milner type system. As far as we know, prior to our work only [24] treats with technical detail a type system for functional logic programming. Our work makes clear contributions when compared to [24]:

- By introducing the notion of critical variables, we are more liberal in the treatment of opaque variables, but still preserving the essential property of subject reduction. Moreover, this liberality extends also to data constructors, dropping the traditional restriction of transparency required to them. This is somehow similar to what happens with *existential types* [49] or *generalized algebraic data types* [57], a connection that we plan to further investigate in the future.
- Our type system considers local pattern bindings and λ-abstractions (also with patterns), that were missing in [24]. In addition to that, we have made a rather exhaustive analysis and formalization of different possibilities for polymorphism in local bindings.
- Subject reduction was proved in [24] wrt. a narrowing calculus. Here we do it wrt. an small-step operational semantics closer to real computations.
- In [24] programs came with explicit type declarations. Here we provide algorithms for inferring types for programs without such declarations, which can easily became part of the type stage of a FL compiler.

Chapter 2

Preliminaries

In this section we will present some definitions and notation used in the rest of the work. We will begin with the syntax of expressions and programs, and definitions about substitutions. Then we will treat types, their syntax and some relations involving them. Finally, we will present sets of assumptions, the way of storing type assumptions over symbols in our framework.

A global writing convention used in this work is that $\overline{e_n}$ is a sequence of n elements $e_1 \dots e_n$, and e_i the $i - th$ element in that sequence.

2.1 Expressions, programs and substitutions

We assume a signature $\Sigma = DC \cup FS$ where DC and FS are two disjoint sets of *data constructor* and *function* symbols resp., all them with an associated arity. We write DC^n and FS^n for the set of constructor and function symbols of arity n. We also assume a numerable set of *data variables* $\mathcal{DV}$. The syntax of the expressions of the language appears in Figure 2.1. Notice that λ-abstractions and *let* expressions support patterns instead of single variables, and there are three different kinds of let expressions: let_m, let_{pm} and let_p; as explained in the previous chapter. We make a distinction between the different expressions. $X\ e_1 \dots e_m\ (m \geq 0)$ are called *flexible* expressions (*variable application* when $m > 0$). On the other hand, *rigid* expressions have the form $h\ e_1 \dots e_m$; they are called *junk* if $h \in CS^n$ and $m > n$, *active* if $h \in FS^n$ and $m \geq n$, and *passive* otherwise. Expressions like $\lambda t_1 \dots t_n.e$ will be usually written as $\lambda \overline{t_n}.e$, and we will write let_* for any type of let expression: let_m, let_{pm} or let_p. As usual, expression application is left associative, so $e_1\ e_2\ e_3$ is equal to $(e_1\ e_2)\ e_3$. We will use the

metavariable h for any symbol in $DC \cup FS$, and s for any symbol in $DC \cup FS \cup \mathcal{DV}$.

Data variables	$\mathcal{DV}$	X,Y,...
Constructor symbol	DC	c
Function symbol	FS	f
Patterns	$Pat \ni t$ ::=	X | c $t_1 \ldots t_n$ if $c \in DC^m$ and $n \leq m$ | f $t_1 \ldots t_n$ if $f \in FS^m$ and $n < m$
First Order pattern	$FOPat \ni fot$ ::=	X | c $fot_1 \ldots fot_n$ if $c \in DC^n$
Higher Order pattern	$HOPat \ni hot$ ::=	h $t_1 \ldots t_m$ if $h \in DC^n \cup FS^n$ and $m < n$ | c $t_1 \ldots t_n$ if $c \in DC^n$ and some $t_i \in HOPat$
Expression	$Expr \ni e$ ::=	X | c | f | (e_1 e_2) | λt.e if t is linear | let_m t = e_1 in e_2 if t is linear | let_{pm} t = e_1 in e_2 if t is linear | let_p t = e_1 in e_2 if t is linear

Figure 2.1: **Syntax of expressions**

Contexts are expressions with exactly one hole, and are defined in Figure 2.2. The application of contexts to expressions (written $\mathcal{C}[e]$) is also defined in the same figure. Notice that context application may capture variables, for example in $(\lambda X.[])X$.

$var(e)$ is the set of all the variables appearing in e. The set of *free variables* (FV) of an expression e is the set of all the variables in e not bounded by any λ-abstraction or let expression. The formal definition is as follows.

Definition 1 (Free variables of an expression)**.**

$$
\begin{aligned}
FV(X) &= \{X\} \\
FV(h) &= \emptyset \quad \textit{if } h \in DC \cup FS \\
FV(e_1 e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\lambda t.e) &= FV(e) \setminus var(t) \\
FV(let_* \; t = e_1 \; in \; e_2) &= FV(e_1) \cup (FV(e_2) \setminus var(t))
\end{aligned}
$$

$Contxt \ni \mathcal{C} ::=$	$[\,]$
	$\mid \mathcal{C}\ e$
	$\mid e\ \mathcal{C}$
	$\mid \lambda t.\mathcal{C}$
	$\mid let_*\ t = \mathcal{C}\ in\ e$
	$\mid let_*\ t = e\ in\ \mathcal{C}$
$[\,]e =$	e
$(\mathcal{C}\ e')[e] =$	$\mathcal{C}[e]\ e'$
$(e'\ \mathcal{C})[e] =$	$e'\ \mathcal{C}[e]$
$(\lambda t.\mathcal{C})[e] =$	$\lambda t.(\mathcal{C}[e])$
$(let_*\ t = \mathcal{C}\ in\ e')[e] =$	$let_*\ t = \mathcal{C}[e]\ in\ e'$
$(let_*\ t = e'\ in\ \mathcal{C})[e] =$	$let_*\ t = e'\ in\ \mathcal{C}[e]$

Figure 2.2: **Syntax of one-hole contexts and its application**

Notice that with the given definition of FV there are not recursive let-bindings in the language since the possible occurrences of a variable X of t in e_1 are not considered bound and therefore refer to a 'different' X. Here we present some examples of free variables of expressions:

Example 5 (Free variables of an expression)**.**

$FV(\ (\lambda X.\lambda Y.[X, Y, true])\ false\ false\ ) = \emptyset$
$FV(\ \lambda X.[X, Y, true]\ ) = \{Y\}$
$FV(\ let_p\ snd\ X =\ snd\ true\ in\ X\ ) = \emptyset$
$FV(\ let_m\ Y = \lambda X.Y\ X\ in\ Y\ 4\ ) = \{Y\}$

Data substitutions $\theta \in \mathcal{S}ubst$ are finite mappings from data variables to expressions, and we write them as $[X_1/e_1, \ldots, X_n/e_n]$. The application of a substitution θ to an expression e is written as $e\theta$ and defined as follows:

Definition 2 (Application of a substitution to an expression)**.**

$$X\theta = \theta(X)$$
$$h\theta = h \quad if\ h \in DC \cup FS$$
$$(e_1 e_2)\theta = e_1\theta\ e_2\theta$$
$$(\lambda t.e)\theta = \lambda(t\theta').e\theta'\theta$$
$$(let_*\ t = e_1\ in\ e_2)\theta = let_*\ t\theta' = e_1\theta\ in\ e_2\theta'\theta$$

$$being\ \{\overline{\alpha_n}\} = var(t) \cap (Dom(\theta) \cup FV(Rng(\theta)))\ and\ \theta' \equiv [\overline{\alpha_n/\beta_n}]\ with\ \overline{\beta_n}\ fresh$$

Application of substitutions does not capture variable, since it renames the bound variables when

there exists the risk of capture, i.e., they appear in $FV(Rng(\theta))$. Notice that renaming is not performed in e_1 of let expressions. In this case, any variable in $FV(e_1)$ appearing in $var(t)$ will not be bound, because it will refer to a 'different' variable. The following example presents some applications of substitution to expressions:

Example 6 (Application of a substitution to an expression)**.**

$(F\ id\ [1,2,3])\ [F/map] = map\ id\ [1,2,3]$
$(\lambda X.X + 1)\ [X/0] = \lambda Y.Y + 1$
$((\lambda X.X + 1)\ X)\ [X/0] = (\lambda Y.Y + 1)\ 0$
$(let_{pm}\ X = 1\ in\ Z + X)\ [Z/1] = let_{pm}\ X = 1\ in\ 1 + X$
$(let_p\ X = 1\ in\ Z + X)\ [Z/X] = let_p\ W = 1\ in\ X + W$
$(let_m\ X = X + X\ in\ X)\ [X/1] = let_m\ Y = 1 + 1\ in\ Y$

We call *domain* of a substitution to the set of variables which are changed by the substitution: $Dom(\theta) = \{X \in \mathcal{DV} | X\theta \neq X\}$. We call *range* of a substitution to the set $Rng(\theta) = \{X\theta | X \in Dom(\theta)\}$. Given two substitutions θ_1 and θ_2, the *composition* of substitutions is another substitution denoted as $\theta_1\theta_2$ and defined as $X\theta_1\theta_2 = (X\theta_1)\theta_2$. The *simultaneous composition* of two substitutions θ_1 and θ_2 is defined only when the domains are disjoint, i.e., when $Dom(\theta_1) \cap Dom(\theta_2) = \emptyset$. In this case, the simultaneous composition of θ_1 and θ_2 is written as $(\theta_1 + \theta_2)$ and is defined as:

$$X(\theta_1 + \theta_2) = \begin{cases} X\theta_1 & \text{if } X \in Dom(\theta_1) \\ X\theta_2 & \text{otherwise} \end{cases}$$

If A is a set of variables, the *restriction* of a substitution θ_1 to A ($\theta_1|_A$) is defined as:

$$X(\theta_1|_A) = \begin{cases} X\theta_1 & \text{if } X \in A \\ X & \text{otherwise} \end{cases}$$

The most important data substitutions used in this work are *pattern-substitutions*, those substitutions such that their range are patterns instead of expressions. We denote this set of substitutions as $\mathcal{PS}ubst = \{\theta \in \mathcal{S}ubst | Rng(\theta) \in Pat\}$.

A *program rule* is defined as $PRule \ni r\ ::= f\ t_1 \ldots t_n \rightarrow e$ where $n \geq 0$, the set of patterns $\{t_1, \ldots, t_n\}$ is linear (i.e., every variable appears only once in all the patterns) and $FV(e) \subseteq \bigcup_i var(t_i)$. Therefore, extra variables (variables that appear in the right-hand side and do not appear in any of the patterns of the left-hand side) are not considered in this work. A *program* $\mathcal{P}$ is a set of zero or more program rules, $Prog \ni \mathcal{P} = \{r_1, \ldots, r_n\}$ with $n \geq 0$. The union of programs is denoted as $\mathcal{P}_1 \cup \mathcal{P}_2$.

2.2 Types

Types are the kind of objects that the type system will relate with expressions. In oder to define them we assume a numerable set of *type variables* $\mathcal{TV}$ and a countable alphabet $\mathcal{TC} = \bigcup_{n\in\mathbb{N}} \mathcal{TC}^n$ of *type constructors*. As data constructors, type constructors have an associated arity. The syntax of types can be found in Figure 2.3. Sometimes we will refer to a simple type τ only as "type", but in these cases the context will make clear the meaning. Usually, a type-scheme $\forall\alpha_1.\forall\alpha_2 \ldots \forall\alpha_n.\tau$ will be written as $\forall\alpha_1, \alpha_2, \ldots, \alpha_n.\tau$ or simply $\forall\overline{\alpha_n}.\tau$. As usual, the arrow $\rightarrow$ is right associative, so $int \rightarrow int \rightarrow int$ is equal to $int \rightarrow (int \rightarrow int)$.

Type variable	$\mathcal{TV}$	$\alpha, \beta, \gamma, \ldots$
Simple type	$SType \ni \tau$ ::=	α
		| C $\tau_1 \ldots \tau_n$ with $C \in \mathcal{TC}^n$
		| $\tau_1 \rightarrow \tau_2$
Type-scheme	$TScheme \ni \sigma$ ::=	τ
		| $\forall\alpha.\sigma$

Figure 2.3: **Syntax of the types**

The set of *free type variables* of a type-scheme is the set of all the type variable not bound by any quantifier. The formal definition is as follows.

Definition 3 (Free type variables of a type-scheme)**.**

$$\begin{aligned} FTV(\alpha) &= \{\alpha\} \\ FTV(C\ \tau_1 \ldots \tau_n) &= \textstyle\bigcup_i FTV(\tau_i) \\ FTV(\tau_1 \rightarrow \tau_2) &= FTV(e_1) \cup FTV(e_2) \\ FTV(\forall\overline{\alpha_n}.\tau) &= FTV(\tau) \setminus \{\overline{\alpha_n}\} \end{aligned}$$

As we have done with data substitutions, we define *type substitutions* as finite mappings from type variables to simple types (not type-schemes): $[\alpha_1/\tau_1, \ldots, \alpha_n/\tau_n]$. These substitutions will be referred as π, and the set of all these substitutions as $\mathcal{TS}ubst$. The application of type substitutions over variables and types is the usual, and over type-schemes is the application only over their free type variables. Based on substitutions, we can define two important relations over type-schemes: *instance* and *generic instance*. The difference between them is that instantation only replaces free type variables by simple types, and generic instantation replaces only the quantified variables, i.e., those understood polymorphically, by simple types. Therefore the generic instances of a type-scheme are all the simple types represented by

that type-scheme.

Definition 4 (Instance of a type-scheme)**.**
We say that σ' is an instance of σ if $\sigma' = \sigma\pi$ for some $\pi \in \mathcal{TS}ubst$.

For example, $bool \to bool$ is an instance of $\alpha \to \alpha$ and $\forall\alpha.\alpha \to int$ is an instance of $\forall\alpha.\alpha \to \beta$.

Definition 5 (Generic instance of a type-scheme σ)**.**
τ is a generic instance of a type-scheme $\sigma \equiv \forall\overline{\alpha_n}.\tau'$ (or σ subsumes τ, or σ is more general than τ) if $\tau \equiv \tau'[\overline{\alpha_n/\tau_n}]$ for some types $\overline{\tau_n}$. We represent that $\sigma \succ \tau$. We extend $\succ$ to a relation between type-schemes by saying that $\sigma \succ \sigma'$ iff for every type τ such that $\sigma' \succ \tau$ then $\sigma \succ \tau$. Notice that with simple types $\tau \succ \tau'$ iff $\tau \equiv \tau'$.

Examples:

- $\forall\alpha.\alpha \to \alpha \succ bool \to bool$
- $\forall\alpha.\alpha \to \alpha \succ \forall\beta.\forall\gamma.(\beta, \gamma) \to (\beta, \gamma)$
- $(\alpha, int) \succ (\alpha, int)$
- $\forall\alpha.\forall\beta.[\alpha] \to [\beta] \to [(\alpha, \beta)] \succ \forall\gamma.[\gamma \to bool] \to [\delta] \to [(\gamma \to bool, \delta)]$

There exists a useful characterization of $\succ$ with a more operational behavior which is explained in [19, 62]. This characterization is presented in the following proposition.

Proposition 1 (Characterization of $\succ$)**.**
A type-scheme $\forall\alpha_1 \ldots \alpha_n.\tau \succ \forall\beta_1 \ldots \beta_m.\tau'$ iff

1.- there is a type substitution $\pi \equiv [\overline{\alpha_n/\tau_n}]$ (for some types $\tau_1 \ldots \tau_n$) such that $\tau' \equiv \tau\pi$

2.- $\beta_1 \ldots \beta_m$ do not occur free in $\forall\alpha_1 \ldots \alpha_n.\tau$

The proof of the correctness of this characterization can be found in [19].

Notice that $\succ$ is a reflexive and transitive relation. Therefore we can define an equivalence relation between type-schemes in the following way.

Definition 6 (Equivalence between type-schemes)**.**
$\sigma \equiv_\succ \sigma' \Longleftrightarrow_{def} \sigma \succ \sigma' \wedge \sigma' \succ \sigma$.

Clearly the relation $\equiv_\succ$ is reflexive, symmetric and transitive. We will usually refer to it simply as $=$. The last useful notion about types is that of *variant of a type-scheme*. This notion is needed when formalizing the inference relation (Chapter 4).

Definition 7 (Variant of a type-scheme)**.**
We say that a type τ' is a variant of a type-scheme $\sigma \equiv \forall\overline{\alpha_n}.\tau$ if $\tau' \equiv \tau[\overline{\alpha_n/\beta_n}]$ with $\overline{\beta_n}$ fresh type variables. We usually write it as $\sigma \succ_{var} \tau'$.

Intuitively, a variant of a type-scheme is a simple type where all the quantified variables have been replaced by variables which have not been used before. Examples of variants are $\forall\alpha.\alpha \to \alpha \succ_{var} \beta \to \beta$ and $\forall\alpha.\alpha \to \beta \to (\alpha, \beta) \succ_{var} \gamma \to \beta \to (\gamma, \beta)$. Now we will present two remarks that come from the previous definitions. The first one is related to the equivalence of α-converted type-schemes, and the second is about the fact that generic instances have more free type variables.

Remark 1.
Note that $\forall\overline{\alpha_n}.\tau \equiv_{\succ} \forall\overline{\beta_n}.\tau[\overline{\alpha_n/\beta_n}]$ if $\{\overline{\beta_n}\} \cap FTV(\tau) = \emptyset$. In other words, two different type-schemes are the same if we change the bounded variables for other variables which do not appear free in τ. For example, $\forall\alpha, \beta.(\alpha, \beta) \to \alpha$ is equal to $\forall\gamma, \delta.(\gamma, \delta) \to \gamma$.

Remark 2.
If $\sigma \succ \sigma'$ then $FTV(\sigma) \subseteq FTV(\sigma')$.
It is clear from the characterization of $\succ$ given in Proposition 1. If α is a type variable in $FTV(\sigma)$ then it will not be affected by π. Besides it will not be generalized, i.e., α will be different from the generalized variables β_j. Therefore $\alpha \in FTV(\sigma')$ and $\alpha \in FTV(\sigma) \Longrightarrow \alpha \in FTV(\sigma')$, so $FTV(\sigma) \subseteq FTV(\sigma')$.

Sets of assumptions are the constructions that relate types-schemes to symbols in our type system. They are not only important in type derivations, but they are key in type inference because they also "store" the type constraints found for every symbol.

Definition 8 (Set of assumptions)**.**
A set of assumptions is a set of the form $\{s_1 : \sigma_1, \ldots, s_n : \sigma_n\}$, where s_i is a function symbol, data constructor symbol or variable and σ_i is a type-scheme. We usually denote a set of assumptions with $\mathcal{A}$. We can view $\mathcal{A}$ as a partial function defined as:

$$\mathcal{A}(s) = \begin{cases} \sigma & \text{if } \{s : \sigma\} \in \mathcal{A} \\ undefined & \text{otherwise} \end{cases}$$

Sets of assumptions can be extended with new assumptions over new or existing symbols. If the symbol does not appear in the original set of assumptions, the new assumption is added; otherwise the previous assumption is discarded and then the new assumption is added.

Definition 9 (Adding an assumption to $\mathcal{A}$)**.**
Let $\mathcal{A}_s$ be the set resulting of discarding all the assumptions over the symbol s in $\mathcal{A}$. Then adding the

assumption $\{s : \sigma\}$ to $\mathcal{A}$ is defined as $\mathcal{A} \oplus \{s : \sigma\} = \mathcal{A}_s \cup \{s : \sigma\}$. This notion is extended to adding sets of assumptions in the usual way: $\mathcal{A} \oplus \{\overline{s_n : \sigma_n}\} = \mathcal{A} \oplus \{s_1 : \sigma_1\} \oplus \ldots \oplus \{s_n : \sigma_n\}$.

$\oplus$ operator is left associative, so $\mathcal{A} \oplus \{s : \sigma\} \oplus \{s' : \sigma'\} = (\mathcal{A} \oplus \{s : \sigma\}) \oplus \{s' : \sigma'\}$. Notice that it does not matter the order when adding assumptions over different symbols: the resulting set of assumptions is the same.

Remark 3.
If $s \neq s'$ then $\mathcal{A} \oplus \{s : \sigma\} \oplus \{s' : \sigma'\}$ is the same as $\mathcal{A} \oplus \{s' : \sigma'\} \oplus \{s : \sigma\}$. This remark can be extended to sets of assumptions, in the sense that $\mathcal{A} \oplus \{\overline{X_n : \sigma_n}\} \oplus \{\overline{X'_m : \sigma'_m}\} = \mathcal{A} \oplus \{\overline{X'_m : \sigma'_m}\} \oplus \{\overline{X_n : \sigma_n}\}$ if $X_i \neq X'_j$ for $1 \leq i \leq n$ and $1 \leq j \leq m$.

The previous notion of free type variables is easily extended to set of assumptions: the free type variables of a set of assumptions $\mathcal{A}$ are the free type variables in all the type-schemes appearing in $\mathcal{A}$. This notion if formally defined as follows:

Definition 10 (Free type variables of $\mathcal{A}$)**.**
$FTV(\{\overline{s_n : \sigma_n}\}) = \bigcup_{i=1}^{n} FTV(\sigma_i)$

To finish this section we will present the important notion of *generalization* of a type to obtain a type-scheme. This operation is used in type derivation and is essential to support polymorphism.

Definition 11 (Generalization of τ wrt. $\mathcal{A}$)**.**
$Gen(\tau, \mathcal{A}) = \forall\alpha_1 \ldots \forall\alpha_n.\tau \; where \; \{\alpha_1, \ldots, \alpha_n\} = FTV(\tau) \setminus FTV(\mathcal{A})$.

Examples:

- $Gen([int], \{true : bool, false : bool, f : \alpha \to \alpha\}) = [int]$
- $Gen(\alpha, \{true : bool, false : bool\}) = \forall\alpha.\alpha$
- $Gen(\alpha, \{true : bool, false : bool, f : \alpha \to \alpha\}) = \alpha$
- $Gen(\alpha \to \beta, \{true : bool, false : bool, f : \alpha \to \alpha\}) = \forall\beta.\alpha \to \beta$

As can be seen, the generalization of a type τ is the type-scheme resulting of quantifying all the type variables not appearing free in $\mathcal{A}$. Then it is clear that it depends only on the free type variables of the set of assumptions, not in the particular set of assumptions involved, as the following remark states.

Remark 4.
If $FTV(\mathcal{A}) = FTV(\mathcal{A}')$ then $Gen(\tau, \mathcal{A}) = Gen(\tau, \mathcal{A}')$

Chapter 3

Type Derivations

Type derivations (or type judgements) allow us to relate a type with an expression. This type can be unique, if the expression is monomorphic; or multiple, if the expression is polymorphic. In this section we present two typing relations based on Damas & Milner type system but handling appropriately HO patterns and the different kinds of local definitions.

3.1 Rules of the type system

We have found convenient to separate the task of giving a regular Damas & Milner type to an expression and the task of checking critical variables, as discussed in Section 1.3.1. The first task is performed by the typing relation $\vdash$, which also handle the different kinds of local definitions; and the second by $\vdash^{\bullet}$. Notice that these relations permit us to derive types for expression, not programs. This is not a problem because as we will see in Section 3.3 the notion of well-typed program will be defined using these relations over expressions.

3.1.1 Basic typing relation $\vdash$

The rules of the basic typing relation $\vdash$ appear in Figure 3.1. All the rules have been designed allowing compound patterns in λ-abstractions and let expressions. This is a feature implemented in existing languages but rarely expressed directly in the formalization of their type systems. We have made the effort of keeping explicitly the treatment of these compound patterns inside the type system, instead of

[ID]	$\dfrac{}{\mathcal{A} \vdash s : \tau}$	if $s \in DC \cup FS \cup \mathcal{DV}$ $\wedge\ \mathcal{A}(s) = \sigma \wedge\ \sigma \succ \tau$
[APP]	$\dfrac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1 e_2 : \tau}$	
[Λ]	$\dfrac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau_t \to \tau}$	if $\{\overline{X_n}\} = var(t)$
[LET$_m$]	$\dfrac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \textbf{let}_m\ t = e_1\ \textbf{in}\ e_2 : \tau_2}$	if $\{\overline{X_n}\} = var(t)$
[LET$^X_{pm}$]	$\dfrac{\mathcal{A} \vdash e_1 : \tau_1 \quad \mathcal{A} \oplus \{X : Gen(\tau_1, \mathcal{A})\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \textbf{let}_{pm}\ X = e_1\ \textbf{in}\ e_2 : \tau_2}$	
[LET$^h_{pm}$]	$\dfrac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash h\ t_1 \ldots t_m : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \textbf{let}_{pm}\ h\ t_1 \ldots t_m = e_1\ \textbf{in}\ e_2 : \tau_2}$	if $\{\overline{X_n}\} = var(t_1 \ldots t_m)$
[LET$_p$]	$\dfrac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : Gen(\tau_n, \mathcal{A})}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \textbf{let}_p\ t = e_1\ \textbf{in}\ e_2 : \tau_2}$	if $\{\overline{X_n}\} = var(t)$

Figure 3.1: **Rules of the type system**

depending on any transformation in order to derive types. This way we wanted to achieve a type system that provides more intuition about the types related to an expression.

The first two rules are very similar to the original Damas & Milner type system. Rule **[ID]** handles data variables, data constructors and function symbol. A valid type for that expressions will be any generic instance of the type-scheme stored in the set of assumptions $\mathcal{A}$. Notice that it is mandatory that there exists an assumption for that symbol in $\mathcal{A}$, otherwise the expression will not have any type at all. Rule **[APP]** types an application of expressions. It states that if you can give a functional type $\tau_1 \rightarrow \tau$ to e_1 and the same type τ_1 to the expression e_2, then the whole application $e_1\ e_2$ will have type τ.

Rule **[**Λ**]** differs from Damas & Milner because it supports compound patterns in the λ-abstraction instead of single variables. In the original type system, you first need to guess a type τ_x for the variable. If you can give some type τ to the body of the λ-abstraction using that assumption over the variable, then the whole expression will have type $\tau_x \rightarrow \tau$. In our type system, the guess is extended to all the variables appearing in the pattern. If with those assumptions over the variables it is possible to give a type τ_t to the whole pattern and a type τ to the body of the λ-abstraction, then the whole expression will have type $\tau_t \rightarrow \tau$. Notice that in the case that the pattern is a single variable the judgment $\mathcal{A} \oplus \{X : \tau_x\} \vdash X : \tau_x$ will always succeed, so the behavior will be the same that the original type system. Notice also that the guessed types are simple types and not type-schemes. This limitation prevents to define a function whose arguments are polymorphic, for example $\lambda F.(F\ true, F\ 0)$. It is impossible to guess a simple type that has the form $bool \rightarrow \tau$ and $int \rightarrow \tau$ at the same time, so the expression does not have any type. It may seem surprising that an expression like $(\lambda F.(F\ true, F\ 0))\ id$ has not any type, because its β-reduction is $(id\ true, id\ 0)$, which does not present any problem. This limitation was considered in the original paper from Milner [48], and was imposed to the type system to be decidable and permit an automatic type inference algorithm. Therefore the type system rejects some expressions that will not present any type error during execution. Although monomorphic λ-abstractions is a common limitation in type systems based on Damas & Milner, some authors have proposed type systems supporting them that also permit automatic type inference. This is the case of HMF [41] and HML [42], type systems created by Leijen from Microsoft Research which support polymorphic λ-abstractions via type annotations in the argument; or the proposal of adding polymorphic abstraction to ML by Kfoury and Wells [40]. In this work we have followed the convention from Milner due to its simplicity and because it is the choice of the most popular functional and functional logic languages.

The last four rules handle local definitions with different kinds of polymorphism. The first one is the simplest, treating all the variables in the pattern monomorphically. Rule **[LET**$_m$**]** guesses simple types for those variables in the pattern, and gives a type for it. Then it gives **the same** type for e_1 using the original set of assumptions $\mathcal{A}$. Notice the use of the original set of assumptions in this step, which

prevents recursive definitions. If a variable X appears in the pattern and in e_1, the type for the second occurrence will come from $\mathcal{A}$, independently of the type guessed in the first step, i.e., the occurrences will refer to different variables X. Finally the set of assumptions extended with the guessed types for the variables is used to give a type for e_2, which will be the type of the whole expression. Rule **[LET$^X_{pm}$]** handle the case of a let_{pm} expression with a single variable as a pattern, the situation in which it will be treated polymorphically. First it gives a simple type τ_1 to e_1, and then it gives a type τ_2 for e_2 using the original $\mathcal{A}$ extended with an assumption that relates the variable with the generalization of τ_1 wrt. $\mathcal{A}$. It is this generalization step (Definition 11) which provides the polymorphism to the variable. The key resides in the fact that if you can give a type for an expression containing type variables which do not appear in the set of assumptions, you can also give to that expression any other type in which those type variables have been replaced by simple types. If we replace X by e_1 in e_2, each occurrence of e_1 could have different types in the previous way. Then the generalization step quantifies the type variables which do not appear free in $\mathcal{A}$, those that could be any type, generating a type-scheme whose generic instances will be all the possible types for e_1. With this assumption, X will behave like e_1 and its potential polymorphism. Rule **[LET$^h_{pm}$]** is the same as **[LET$_m$]** because let_{pm} treats compound patterns monomorphically. Finally, rule **[LET$_p$]** behaves like **[LET$^X_{pm}$]** but extends the polymorphism to all the variables in the pattern. It needs an additional step to check that the guessed types for the variables give a valid type to pattern that is the same as the type for e_1. Notice that in this case the generalization may lose the connection between the guessed types for the variables. This can be seen in the expression $let_p\ [F, G] = [id, id]\ in\ (F\ 0, G\ true)$, whose type derivation appears in Example 7-*4)*. The type for both F and G can be $\alpha \to \alpha$ (being α a variable not appearing in $\mathcal{A}$), but the generalization step will assign both the type-scheme $\forall\alpha.\alpha \to \alpha$. This will allow us to derive a type $int \to int$ for F and $bool \to bool$ for G, yielding a type $(int, bool)$ for the whole expression. The difference between the two types might seem surprising, because both F and G appear in the same list, but it can be easily understood if we see the pattern matching as projection functions. The list $[id, id]$ has the polymorphic type $\forall\alpha.[\alpha \to \alpha]$. When we project the first element we can assume that $[id, id]$ has type $[int \to int]$, so F has type $int \to int$. But when we project the second element, we can freely assume that $[id, id]$ has type $[bool \to bool]$, because it is polymorphic, and that is why G can have type $bool \to bool$.

Here we show some examples of type derivations:

Example 7 (Type derivations)**.**
1) *This is the λ-abstraction associated to co, the problematic function in Example 3. It has a valid type using the basic typing relation but we can see the origin of the problems in its returned value, which has type γ independently of the type of the argument. Let $\mathcal{A}_1$ be the set of assumptions $\{snd : \forall\alpha, \beta.\alpha \to \beta \to \beta\}$*

$$\textit{[}\Lambda\textit{]}\ \frac{\textit{[APP]}\ \dfrac{\textit{[ID]}\ \dfrac{}{\mathcal{A}_1 \oplus \{X:\gamma\} \vdash snd : \gamma \to \delta \to \delta} \quad \textit{[ID]}\ \dfrac{}{\mathcal{A}_1 \oplus \{X:\gamma\} \vdash X : \gamma}}{\mathcal{A}_1 \oplus \{X:\gamma\} \vdash snd\ X : \delta \to \delta} \quad \textit{[ID]}\ \dfrac{}{\mathcal{A}_1 \oplus \{X:\gamma\} \vdash X : \gamma}}{\mathcal{A}_1 \vdash \lambda snd\ X.X : (\delta \to \delta) \to \gamma}$$

2) *The following is an example of completely monomorphic let.* F *has type* $int \to int$*, so it can only be applied to expressions of type* int*. Let* $\mathcal{A}_2$ *be the set of assumptions* $\{id : \forall\alpha.\alpha \to \alpha, 0 : int\}$

$$\textit{[LET}_m\textit{]}\ \frac{\textbf{[ID]}\ \dfrac{}{\mathcal{A}_2 \oplus \{F : int \to int\} \vdash F : int \to int} \quad \textit{[ID]}\ \dfrac{}{\mathcal{A}_2 \vdash id : int \to int} \quad \textit{[APP]}\ \dfrac{(\ldots)}{\mathcal{A}_2 \oplus \{F : int \to int\} \vdash F\ 0 : int}}{\mathcal{A}_2 \vdash let_m\ F = id\ in\ F\ 0 : int}$$

3) *The next example shows the polymorphism of* let_{pm} *expressions when the pattern is a variable. Here* F *has a polymorphic type* $\forall\gamma.\gamma \to \gamma$ *when deriving the type of* $(F\ 0, F\ true)$*, so it can be used with types* $int \to int$ *and* $bool \to bool$*. Let* $\mathcal{A}_3$ *be the set of assumptions* $\{id : \forall\alpha.\alpha \to \alpha, 0 : int, true : bool, ()^1 : \forall\alpha,\beta.\alpha \to \beta \to (\alpha,\beta)\}$

$$\textit{[LET}^X_{pm}\textit{]}\ \frac{\textit{[ID]}\ \dfrac{}{\mathcal{A}_3 \vdash id : \gamma \to \gamma} \quad \textit{[APP]}\ \dfrac{(\ldots)}{\mathcal{A}_3 \oplus \{F : \forall\gamma.\gamma \to \gamma\} \vdash (F\ 0, F\ true) : (int, bool)}}{\mathcal{A}_3 \vdash let_{pm}\ F = id\ in\ (F\ 0, F\ true) : (int, bool)}$$

4) *The following example is very similar to the previous one but using a* let_p *expression. It shows how variables that occur in the same list pattern are used with different types in the body of the let expressions. Let* $\mathcal{A}_4$ *be the set of assumptions* $\{id : \forall\alpha.\alpha \to \alpha, 0 : int, true : bool, ()^1 : \forall\alpha,\beta.\alpha \to \beta \to (\alpha,\beta), [\,] : \forall\alpha.[\alpha], cons : \forall\alpha \to [\alpha] \to [\alpha]\}$

$$\textit{[LET}_p\textit{]}\ \frac{\textbf{[APP]}\ \dfrac{(\ldots)}{\mathcal{A}_4 \oplus \{F : \gamma \to \gamma, G : \gamma \to \gamma\} \vdash [F,G] : [\gamma \to \gamma]} \quad \textit{[APP]}\ \dfrac{(\ldots)}{\mathcal{A}_4 \vdash [id.id] : [\gamma \to \gamma]} \quad \textit{[APP]}\ \dfrac{(\ldots)}{\mathcal{A}'_4 \vdash (F\ 0, G\ true) : (int, bool)}}{\mathcal{A}_4 \vdash let_p\ [F,G] = [id,id]\ in\ (F\ 0, G\ true) : (int, bool)}$$

being $\mathcal{A}'_4 \equiv \mathcal{A}_4 \oplus \{F : \forall\gamma.\gamma \to \gamma, G : \forall\gamma.\gamma \to \gamma\}$

Other of our aims when designing the type system has been to obtain syntax-directed type judgments. In the original Damas & Milner type system [20] appear two rules, *INST* and *GEN*, that can occur anywhere in the type judgement. The type system relates expressions with type-schemes, and these rules allow to obtain generic instances and more general type-schemes from previous ones. Since these rules can appear anywhere in the type derivation, it does not depend on the form of the expression. To solve this problem we will eliminate these two rules and integrate into the others. As [62, 17] say, limiting the type relation to simple types instead of type-schemes, any derivation can be build using *GEN* just before the rule of LET (to obtain a general type for the variables) and *INST* before the rule for the symbols (to obtain a generic instance). Therefore we have integrated the instantation step into the rule **[ID]**,

[1] tuple constructor of arity 2

which gives generic instances only; and the generalization step in the rules **[LET**$^X_{pm}$**]** and **[LET**$_p$**]**, those supporting polymorphism. This way given an expression there is only one typing rule that can be used, so the form of the type derivation depends only on the syntax of the expression.

3.1.2 Extended typing relation $\vdash^\bullet$

The previous basic type relation $\vdash$ gives a type to an expression, handling the different kinds of polymorphism in let expressions and the occurrences of compound patterns instead of single variables. In this section we introduce the extended typing relation $\vdash^\bullet$ that uses the previous one to give types to expressions but also enforces the absence of critical variables, i.e., opaque variables which are "used" in the expression. As we have seen in Section 1, the intuition behind an *opaque variable* is that is a data variable of a pattern in a λ-abstraction or let expression whose type is not univocally fixed by the type of the pattern. The problem arises in that due to this opacity we can build type derivations which assigns incorrect assumptions to the variables. Let us illustrate this by an example:

Example 8.
Let $\mathcal{A}$ be the set of assumptions $\{snd : \forall\alpha,\beta.\alpha \to \beta \to \beta, true : bool, 0 : int\}$ and let e be the expression $let_p\ snd\ X = snd\ true\ in\ X$. A possible type derivation for this expressions is:

$$\textbf{[LET}_p\textbf{]}\ \frac{\begin{array}{c}\mathcal{A} \oplus \{X : int\} \vdash snd\ X : int \to int \\ \mathcal{A} \vdash snd\ true : int \to int \\ \mathcal{A} \oplus \{X : int\} \vdash X : int\end{array}}{\mathcal{A} \vdash let_p\ snd\ X = snd\ true\ in\ X : int}$$

The problem here is since X is opaque, the fact that $snd\ X$ and $snd\ true$ have the same type does not provide any information about that the assumption $\{X : int\}$ is correct, and we have the false impression that it is a valid assumption. Instead of int we could have given to X any other type, and we would have obtain an expression that returns $true$ with any type. A similar situation appears in λ-abstractions.

In [24] it is assumed that expressions contain only *transparent patterns*. In this framework, type signatures for constructors and functions are part of the signature of the program. A type $\overline{\tau_m} \to \tau$ is called *m-transparent* if $FTV(\overline{\tau_m}) \subseteq FTV(\tau)$, and a function f is m-transparent if its type is m-transparent. Then, a pattern $h\ \overline{t_n}$ (with $h \in DC \cup FS$) is called transparent if h is n-transparent and $\overline{t_n}$ are also transparent patterns; and opaque otherwise. In this framework patterns as $snd\,X$ or $snd\,true$ are opaque, because in both cases snd is applied to one expression, and snd is not 1-transparent, so they are not valid patterns in the left-hand side of the rules. We have found that this restriction can be relaxed, because an opaque pattern can contain subpatterns whose type is actually fixed by the type of the whole pattern. The problem with opaque variables arises when they are "used" in the expression, as in the previous

example. But if a variable is not "used" in the body, for example in $let_p\ snd\ X = snd\ true\ in\ true$, the assumption will not be used in the type derivation. Then, only the occurrence of opaque variables that are "used" in the expression (i.e., are critical) must be checked.

In order to introduce the extended typing relation $\vdash^\bullet$ we need first to formally define opaque and critical variables. The notion of *opaque variable of a pattern* relies on the typing relation $\vdash$, as the following definition states:

Definition 12 (Opaque variable of t wrt. $\mathcal{A}$)**.**
Let t be a pattern that admits type wrt. a given set of assumptions $\mathcal{A}$. We say that $X_i \in \{\overline{X_n}\} = var(t)$ is opaque wrt. $\mathcal{A}$ iff $\exists\overline{\tau_n}, \tau$ s.t. $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$ and $FTV(\tau_i) \nsubseteq FTV(\tau)$.

Examples:

- *X is an opaque variable in $snd\ X$ when $\mathcal{A}$ contains the usual type assumption for $snd : \forall\alpha \to \beta \to \beta$). In this case we can build the type derivation $\mathcal{A} \oplus \{X : \gamma\} \vdash snd\ X : \delta \to \delta$ and clearly $FV(\gamma) \nsubseteq FV(\delta \to \delta)$.*
- *The tuple pattern (X, Y) does not have any opaque variable, assuming the usual type for the tuple constructor. In this case it is not possible to create a type derivation in which the types of the variables X and Y do not appear in the type of the whole pattern.*
- *X is not opaque in $snd\ [X, true]$ according to the usual type assumptions for snd and list constructors. Although the type of the argument of snd does not appear in the type of the whole pattern, the type of X is fixed to bool by the subpattern $[X, true]$ where it appears.*

Since it is based on the existence of a certain type derivation, Definition 12 cannot be used as an effective check for the opacity of variables. But it is possible to exploit the close relationship between $\vdash$ and type inference $\Vdash$ that will be presented in Chapter 4. Since $\Vdash$ can be viewed as an algorithm, Proposition 2 provides a more operational characterization of opaque variable which is useful when implementing the type system.

Proposition 2 (Alternative characterization of opaque variable)**.**
Let t be a pattern that admits type wrt. a given set of assumptions $\mathcal{A}$. We say that $X_i \in \{\overline{X_n}\} = var(t)$ is opaque wrt. $\mathcal{A}$ iff $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_g | \pi_g$ and $FTV(\alpha_i\pi_g) \nsubseteq FTV(\tau_g)$.

The proof of the equivalence of Definition 12 and Proposition 2 can be found in Lemma 4 of Appendix A. As we have seen, not all opaque variables are problematic, but only those that are "used" in the expression. In this work we consider the simplest notion of "use": a variable is used if it occurs in the body of a λ-abstraction or let expression. More complex notions can be developed, based on static

analysis of the program rules. For example given the program $\{const\ X\ \rightarrow\ 0\}$ and the expression $let_p\ snd\ X = snd\ true\ in\ const\ X$ the variable X will not be used, even though it occurs in the body of the let expression. In this case there will not be problematic that opacity avoids us to know the type of X, because no pattern matching will be performed with it. Notice that knowing if a matched variable will be effectively used in a computation is an undecidable problem, so an approximation must be chosen.

Once that we have a notion of "used variable", we can define formally the notion of *critical variables* of an expression wrt. a set of assumptions. In the following definition we present critical variables considering our simple notion of "used variable". For more complex notions the definition is the same but replacing $FV(..)$ by the particular check.

Definition 13 (Critical variables of e wrt. $\mathcal{A}$)**.**

$$critVar_{\mathcal{A}}(s) = \emptyset$$
$$critVar_{\mathcal{A}}(e_1 e_2) = critVar_{\mathcal{A}}(e_1) \cup critVar_{\mathcal{A}}(e_2)$$
$$critVar_{\mathcal{A}}(\lambda t.e) = (opaqueVar_{\mathcal{A}}(t) \cap FV(e)) \cup critVar_{\mathcal{A}}(e)$$
$$critVar_{\mathcal{A}}(let_*\ t = e_1\ in\ e_2) = (opaqueVar_{\mathcal{A}}(t) \cap FV(e_2)) \cup\ critVar_{\mathcal{A}}(e_1) \cup critVar_{\mathcal{A}}(e_2)$$

Examples:

- *As we have seen in the examples of the definition of opaque variables (Definition 12), X is opaque in $snd\ X$. Therefore $critVar_{\mathcal{A}}(\lambda snd\ X.X) = \{X\}$ because X appears in the body of the λ-abstraction.*
- *Although X is opaque in $snd\ X$, $critVar_{\mathcal{A}}(let_m\ snd\ X = snd\ true\ in\ 0) = \emptyset$ because X does not occur in the rest of the expression.*
- *As we have seen before X is not opaque in $snd\ [X, true]$, so trivially $critVar_{\mathcal{A}}(\lambda snd\ [X, true].X) = \emptyset$.*

With the previous definitions we can now formalize the extended typing relation. It only contains one rule, **[P]** (Figure 3.2). This typing relation uses the basic one to give a type to an expression, and then checks the absence of critical variables. Here we show some examples of type derivations with the

$$\textbf{[P]}\quad \frac{\mathcal{A} \vdash e : \tau}{\mathcal{A} \vdash^{\bullet} e : \tau}\quad \text{if } critVar_{\mathcal{A}}(e) = \emptyset$$

Figure 3.2: **Rule of the extended type system**

extended relation $\vdash^{\bullet}$:

Example 9 (Extended type derivations)**.**

1. *Let* $\mathcal{A}_1$ *be* $\{snd : \forall\alpha, \beta.\alpha \to \beta \to \beta\}$*. Then* $\mathcal{A}_1 \not\vdash^\bullet \lambda snd\ X.X$ *because although* $\mathcal{A}_1 \vdash \lambda snd\ X.X : (\delta \to \delta) \to \gamma$ *(from Example 7-1) we have seen that* $critVar_{\mathcal{A}_1}(\lambda snd\ X.X) = \{X\}$*.*

2. *Using* $\mathcal{A}_2 \equiv \{snd : \forall\alpha, \beta.\alpha \to \beta \to \beta, true : bool\}$*,* $\mathcal{A}_2 \vdash^\bullet \lambda snd\ X.true : (\delta \to \delta) \to bool$ *because* $\mathcal{A}_2 \vdash \lambda snd\ X.true : (\delta \to \delta) \to bool$ *and* X *is not critical in the expression.*

3. $\mathcal{A}_3 \vdash^\bullet let_{pm}\ F = id\ in\ (F\ 0, F\ true) : (int, bool)$*, since* $\mathcal{A}_3 \vdash let_{pm}\ F = id\ in\ (F\ 0, F\ true) : (int, bool)$ *(from Example 7-3) and there are not critical variables in the expression.*

3.2 Properties of type derivations

The typing relations fulfill a set of useful properties. Here we will use $\vdash^?$ for any of the two typing relations: $\vdash$ or $\vdash^\bullet$.

Theorem 1 (Properties of the typing relations)**.**

a) If $\mathcal{A} \vdash^? e : \tau$ *then* $\mathcal{A}\pi \vdash^? e : \tau\pi$*, for any* $\pi \in \mathcal{TS}ubst$

b) Let s *be a symbol not appearing in* e*. Then* $\mathcal{A} \vdash^? e : \tau \iff \mathcal{A} \oplus \{s : \sigma_s\} \vdash^? e : \tau$*.*

c) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e : \tau$ *and* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e' : \tau_x$ *then* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e[X/e'] : \tau$*.*

d) If $\mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$ *and* $\sigma' \succ \sigma$*, then* $\mathcal{A} \oplus \{s : \sigma'\} \vdash e : \tau$*.*

Part *a)* states that type derivations are closed under type substitutions, i.e., from an instance of the set of assumptions we can derive a type that is an instance of the original. *b)* shows that type derivations for e depend only on the assumptions for the symbols in e. Then we can add or delete assumptions over symbols not appearing in the expression and the type derivations will remain valid. *c)* is a substitution lemma stating that in a type derivation we can replace a variable by an expression with the same type. Finally, *d)* establishes that from a valid type derivation we can change the assumption of a symbol for a more general type-scheme, and we still have a correct type derivation for the same type. Notice that this is not true wrt. the typing relation $\vdash^\bullet$ because a more general type can introduce opacity. For example the variable X is not opaque in $snd\ X$ with the type $bool \to bool \to bool$ for snd, but with a more general type such as $\forall\alpha, \beta.\alpha \to \beta \to \beta\ X$ will be opaque.

3.3 Subject reduction

Subject reduction is a key property for type systems, meaning that evaluation does not change the type of an expression. This ensures that run-time type errors will not occur. Examples of this kind of errors are adding two boolean expressions ($true + true$) or applying the logical and function to integers ($and\ 0\ 0$). Notice that partial functions can produce pattern matching errors during execution. An example of this is the function $head$, which returns the first element of a list. This function is usually defined only for non empty lists. Applying $head$ to an empty list [] will usually yield to a pattern matching error, but this is not considered a run-time type error, so the subject reduction property does not assures the absence of this kind of run-time errors. Although in essence both errors are the same, calling a function with an argument such that there is not any matching rule, the cause is different. In the former cases the arguments have a different type that the expected, so it will be impossible to find a rule to reduce the expression. In the latter case, the impossibility resides in the partiality of the function, but the arguments have the correct type.

To state a subject reduction property we need two ingredients. First, a way of recognize those program that are consistent with the type system. Second, a notion of evaluation to reduce expressions, i.e., a semantics. We will also need a transformation to adapt the syntax of our expressions to the semantics. All these parts will be explained in the next sections.

3.3.1 Notion of well-typed program

Subject reduction is only guaranteed for *well-typed* programs, i.e., programs that are consistent with the type system.

Definition 14. *[Well-typed rule]*
A program rule $f\ t_1 \ldots t_n \to e$ is well-typed wrt. $\mathcal{A}$ if $\mathcal{A} \vdash^\bullet \lambda t_1 \ldots \lambda t_n.e : \tau$ and τ is a variant of $\mathcal{A}(f)$.

Definition 15. *[Well-typed program]*
A program $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$ if all its rules are well-typed wrt. $\mathcal{A}$. If $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$ we write $wt_{\mathcal{A}}(\mathcal{P})$.

Notice the use of the extended typing relation $\vdash^\bullet$ in the definition of well-typed rule. This is essential to guarantee subject reduction, as we will explain later. The use of $\vdash^\bullet$ prevents us to consider the problematic program in Example 3 as well-typed with the usual assumptions. Since $\lambda snd\ X.X$ does not have any type wrt. $\vdash^\bullet$ (as can be seen in Example 9-1), the rule of co is not well-typed. Although

the restriction that the type of the lambda abstraction associated to a rule must be a variant of the type of the function symbol (and not an instance) might seem strange, it is necessary. Otherwise, the fact that a program is well-typed will not give us important information about functions like the type of their arguments/result, or the number of arguments; and it will make us to consider as well-typed undesirable programs. These situations are illustrated in Examples 10 and 11.

Example 10.
Let $\mathcal{P}$ be the program $\{f\ true \to true, f\ 0 \to true\}$ and $\mathcal{A}$ be the set of assumptions $\{true : bool,\ 0 : int; f : \forall\alpha.\alpha \to bool\}$. With the relaxed restriction we could affirm that $\mathcal{P}$ is well typed wrt $\mathcal{A}$, which is contrary to our intention and intuition. With parametric polymorphism the meaning of the assumption $f : \forall\alpha.\alpha \to bool$ is that f is a function whose argument can have **any** *type and whose result is of type $bool$. Here this is false because f cannot accept any kind of argument (in contrast with the parametric function id), but only $bool$ or int.*

Then the restriction assures that if in a rule an argument (or the result) is of a certain type, in all the other rules this argument must have the same type.

Example 11.
Let $\mathcal{P}$ be the program $\{fst\ X\ Y \to X\}$ and $\mathcal{A}$ be the set of assumptions $\{f : \forall\alpha, \beta.\alpha \to \beta\}$. If we relax the mentioned restriction we could affirm that $wt_{\mathcal{A}}(\mathcal{P})$ because $\mathcal{A} \vdash \lambda X.\lambda Y.X : \gamma \to \delta \to \gamma$, and this type is a generic instance of $\forall\alpha, \beta.\alpha \to \beta$. Looking at the assumptions, we expect f to have only one argument, which is not true. Using the original restriction, if we want $\mathcal{A}$ to make $\mathcal{P}$ well typed, it will need to contain the assumption $\{f : \forall\alpha, \beta.\alpha \to \beta \to \alpha\}$ (or any generic instance), which indicates the number of arguments of the function f.

Furthermore, relaxing this restriction every program with rules that admit any type will be considered well-typed using the assumption $\forall\alpha.\alpha$ for all its function symbols. A final reason supporting the restriction of the variant can be seen in the next example. Although it is somehow artificial, it shows that subject reduction cannot be guaranteed in well-typed programs without the mentioned restriction.

Example 12.
Suppose the program $\mathcal{P} \equiv \{f\ X \to g\ X\}$ and the set of assumptions $\mathcal{A} \equiv \{f : \forall\alpha.\alpha \to \alpha, g : \beta \to \beta, true : bool\}$. If the definition of well-typed program is relaxed to not check that the types of the rules are variants but generic instances, $\mathcal{P}$ will be well-typed wrt. $\mathcal{A}$ because $\lambda X.g\ X$ has type $\beta \to \beta$, which is a generic instance of the type of $f : \forall\alpha.\alpha \to \alpha$. The expression $f\ true$ has type $bool$, but if we use this well-typed program to reduce it to $g\ true$ the obtained expression is ill-typed, since $true$ has not type β.

Although not explicitly stated, in [24] the same restriction is implicitly considered in the definition of a *well-typed* defining rule for a function. This restriction is achieved by forcing the arguments $\overline{t_n}$ and

the result r in the rule $f\ \overline{t_n} \to r$ to have the same type (τ_i and τ respectively) that appear in the declared type of $f : \overline{\tau_n} \to \tau$. These declared types have not quantification, but they are understood as universally quantified, so they establish the same restriction that the variants in our case.

3.3.2 Semantics

In this work we consider the *let*-rewriting relation of [44]. This is strongly equivalent to HOCRWL [23], a previously existing semantic framework for programs with HO non-deterministic functions. But it has an important advantage: unlike HOCRWL, *let*-rewriting provides an operational behavior, representing computations in a more natural way. In this work we have made two minor changes to the original *let*-rewriting so it fulfils our needs. The modified version can be found in Figure 3.3. First we have added type annotations to let expressions. Since *let*-rewriting only support let expression with variables as patterns, let_{pm} is useless because it has the same behavior as let_p. Then in the modified version only appear let_m and let_p expressions. In the **(LetIn)** rule we decided to introduce a let_m instead of a let_p, but it makes no difference as Lemma 12 in Appendix A states. Second, we have split the original **(Flat)** rule into two: **(Flat$_m$)** and **(Flat$_p$)**. Although both behave equal to the original from the point of view of values, the splitting is needed to guarantee type preservation. *let*-rewriting steps are written $\mathcal{P} \vdash e \to^l e'$, or simply $e \to^l e'$ when the program is implicit.

Rule **(Fapp)** uses a program to reduce a function application. This reduction will only happen when the arguments are patterns (notice that θ is a pattern substitution), otherwise the semantics of call-time choice [36, 25] will be violated. When the arguments of applications are non-patterns, **(LetIn)** moves them into local bindings. If the binding of a let expression is a pattern, **(Bind)** applies the substitution. This does not cause any problems because patterns cannot be further rewritten. Useless bindings are erased with **(Elim)**. **(Flat$_*$)** and **(LetAp)** are needed to avoid some reductions to get stuck. Finally, rule **(Contx)** allows to apply any of the previous rules to a subexpression. A more detailed explanation of the rules appear in [44].

As we have noted before, *let*-rewriting does not support let expressions with compound patterns but our syntax does. Instead of making major changes to *let*-rewriting we have chosen to create a transformation from expressions with compound patterns to expressions with variables as patterns. This transformation will be explained in the next section.

Notice also the absence of λ-abstractions in the *let*-rewriting rules. Although λ-abstractions are a very popular in functional languages, they do not have a clear semantics in functional logic languages yet. Despite of this fact, we have decided to include λ-abstractions in our type systems because they are

very useful when defining well-typed rules.

(Fapp) $f\ t_1\theta \ldots t_n\theta \ \rightarrow^l \ r\theta, \quad$ if $(f\ t_1 \ldots t_n \rightarrow r) \in \mathcal{P}$ and $\theta \in \mathcal{PS}ubst$

(LetIn) $e_1\ e_2 \rightarrow^l let_m\ X = e_2\ in\ e_1\ X, \quad$ if e_2 is an active expression, variable application, junk or *let* rooted expression, for X fresh.

(Bind) $let_K\ X = t\ in\ e\ \rightarrow^l\ e[X/t], \quad$ if $t \in Pat$

(Elim) $let_K\ X = e_1\ in\ e_2 \rightarrow^l e_2, \quad$ if $X \notin FV(e_2)$

(Flat$_m$) $let_m\ X = (let_K\ Y = e_1\ in\ e_2)\ in\ e_3\ \rightarrow^l\ let_K\ Y = e_1\ in\ (let_m\ X = e_2\ in\ e_3)$ if $Y \notin FV(e_3)$

(Flat$_p$) $let_p\ X = (let_K\ Y = e_1\ in\ e_2)\ in\ e_3\ \rightarrow^l\ let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3)$ if $Y \notin FV(e_3)$

(LetAp) $(let_K\ X = e_1\ in\ e_2)\ e_3 \rightarrow^l let_K\ X = e_1\ in\ e_2\ e_3, \quad$ if $X \notin FV(e_3)$

(Contx) $\mathcal{C}[e] \rightarrow^l \mathcal{C}[e']$, if $\mathcal{C} \neq [\,]$, $e \rightarrow^l e'$ using any of the previous rules

where $K \in \{m, p\}$

Figure 3.3: **Higher order *let*-rewriting relation $\rightarrow^l$**

3.3.3 Transformation to simple patterns

As we have said, we need a transformation to eliminate compound patterns in order to adapt our syntax to the semantics we are using. There are various ways to perform this transformation, which differ in the strictness of pattern matching. One possibility is to force pattern matching even if no variable of the pattern is needed. In this situation, $let\ [X] = [1]\ in\ true$ will succeed returning $true$ because the pattern matching succeeds, but $let\ [X] = 0\ in\ true$ will fail because $[X]$ cannot match with 0, even though X is not needed in the body. Another possibility is to force the complete matching only when any variable is used. Then the previous example will succeed returning $true$, but $let\ [X] = [1, 2]\ in\ X$ will fail because X is used in the body and $[X]$ and $[1, 2]$ do not match. The last possibility is to match only the parts of the pattern that are needed. Then $let\ [X] = [1, 2]\ in\ X$ will succeed because although $cons\ X\ nil$ does not match with $cons\ 1\ (cons\ 2\ nil)$ (using $cons$ notation of lists) the matching for the first argument of $cons$ is possible. In this work we have chosen the second alternative, which is explained in [53]. This transformation has been enriched with the different kinds of let expressions (Figure 3.4) in order to preserve the types, as is stated in Theorem 2. Here we show some examples of transformation of expressions:

$$
\begin{array}{rl}
TRL(s) = & s, \text{ if } s \in DC \cup FS \cup \mathcal{DV} \\
TRL(e_1\ e_2) = & TRL(e_1)\ TRL(e_2) \\
TRL(let_K\ X = e_1\ in\ e_2) = & let_K\ X = TRL(e_1)\ in\ TRL(e_2), \text{ with } K \in \{m, p\} \\
TRL(let_{pm}\ X = e_1\ in\ e_2) = & let_p\ X = TRL(e_1)\ in\ TRL(e_2) \\
TRL(let_m\ t = e_1\ in\ e_2) = & let_m\ Y = TRL(e_1)\ in\ \overline{let_m\ X_n = f_{X_n}\ Y}\ in\ TRL(e_2) \\
TRL(let_{pm}\ t = e_1\ in\ e_2) = & let_m\ Y = TRL(e_1)\ in\ \overline{let_m\ X_n = f_{X_n}\ Y}\ in\ TRL(e_2) \\
TRL(let_p\ t = e_1\ in\ e_2) = & let_p\ Y = TRL(e_1)\ in\ \overline{let_p\ X_n = f_{X_n}\ Y}\ in\ TRL(e_2)
\end{array}
$$

for $\{\overline{X_n}\} = var(t) \cap var(e_2)$, $f_{X_i} \in FS^1$ fresh defined by the rule $f_{X_i}\ t \to X_i$,
$Y \in \mathcal{DV}$ fresh, t a non variable pattern and t' any pattern.

Figure 3.4: **Transformation rules of let expressions with patterns**

Example 13 (Transformation $TRL(e)$)**.**

1. $TRL(\ id\ (let_m\ X = true\ in\ X)\ ) = id\ (let_m\ X = true\ in\ X)$
2. $TRL(\ (let_{pm}\ F = id\ in\ F)\ 0) = (let_p\ F = id\ in\ F)\ 0$
3. $TRL(let_{pm}\ [X, Y] = [1, 2]\ in\ X\ ) = let_m\ Z = [1, 2]\ in\ let_m\ X = f_X\ Z\ in\ X$
   where f_X is defined with the rule $f_X\ [X, Y] \to X$. Notice that the projection function for Y is not necessary, since it does not appear in the body of the let expression.
4. $TRL(\ let_p\ (X, Y) = true\ in\ 0\ ) = let_p\ Z = true\ in\ 0$
   In this case no projection function is needed. Notice that the binding $(X, Y) = true$ does not appear in the resulting expression, so it will not be performed.

The transformation $TRL(e)$ defined in Figure 3.4 only changes let expressions. If the pattern is a variable it does not change it, except when it is a let_{pm}, that it is replaced by let_p. If the pattern is compound it is replaced by a chain of simple let expressions. First the expression e_1 of the binding is shared in the variable Y. Then the chain of let expressions use projection functions to "extract" the different variables of the pattern, one by one, using the variable Y bound to e_1. This way if no variable is needed then no matching will be performed. Moreover, no rewriting will happen in e_1. It is important to preserve the polymorphism information of let expressions, and reflect it in the transformed expressions. In let_m and let_{pm}, since the variables are monomorphic, the resulting projected variables are also monomorphic. In the case of let_p the resulting projected variables are polymorphic. Notice that the result of the transformation only has let_m or let_p expressions, since without compound patterns let_{pm} is the same as let_p.

Theorem 2 (Type preservation of the let transformation)**.**
Assume $\mathcal{A} \vdash^\bullet e : \tau$ and let $\mathcal{P} \equiv \{\overline{f_{X_n}\ t_n \to X_n}\}$ be the rules of the projection functions needed in the

transformation of e according to Figure 3.4. Let also $\mathcal{A}'$ be the set of assumptions over that functions, defined as $\mathcal{A}' \equiv \{\overline{f_{X_n} : Gen(\tau_{X_n}, \mathcal{A})}\}$, where $\mathcal{A} \Vdash^{\bullet} \lambda t_i.X_i : \tau_{X_i} | \pi_{X_i}$. Then $\mathcal{A} \oplus \mathcal{A}' \vdash^{\bullet} TRL(e) : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.

Apart from type preservation, Theorem 2 also states that the projection functions are well-typed. Then if we start from a well-typed program $\mathcal{P}$ wrt. $\mathcal{A}$ and apply the transformation to all its rules, the program extended with the projections rules will be well-typed wrt. the extended assumptions: $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P} \uplus \mathcal{P}')$. This result is straightforward, because $\mathcal{A}'$ does not contain any assumption for the symbols in $\mathcal{P}$, so $wt_{\mathcal{A}}(\mathcal{P})$ implies $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.

3.3.4 Subject reduction property

With the previous definitions we can now state the subject reduction property. This property holds only for programs without λ-abstractions and whose let expressions are let_m or let_p with variables as patterns. Theorem 3 states that if a *let*-rewriting step is done using a well-typed program, the resulting expression have the same type that the original. It only express the property for a single step, but its extension to any number of steps is trivial.

Theorem 3 (Subject Reduction)**.**
If $\mathcal{A} \vdash^{\bullet} e : \tau$ and $wt_{\mathcal{A}}(\mathcal{P})$ and $\mathcal{P} \vdash e \rightarrow^{l} e'$ then $\mathcal{A} \vdash^{\bullet} e' : \tau$.

For this result to hold it is essential that the definition of well-typed program relies on $\vdash^{\bullet}$, because it ensures the absence of critical variables in the rules and therefore the reduction of function calls will not create problems with the types. A counterexample can be found in Example 3 from Section 1. The program would be well-typed wrt. $\vdash$, because the λ-abstractions associated to the rules have valid types, and they will be consistent with the usual assumptions $\mathcal{A} \equiv \{snd : \forall\alpha, \beta.\alpha \rightarrow \beta \rightarrow \beta, and : bool \rightarrow bool \rightarrow bool, co : \forall\alpha, \beta.(\alpha \rightarrow \alpha) \rightarrow \beta, cast : \forall\alpha, \beta.\alpha \rightarrow \beta, true : bool, 0 : int\}$. Here, the cause of the problems is the function co. It has a variable X that is opaque in $snd\ X$ and appears in the right-hand side of the rule, i.e., is critical. Then the expression $and\ (cast\ 0)\ true$ has type $bool$, but performing a *let*-rewriting step we obtain $and\ 0\ true$, which has no type at all.

The proof of the subject reduction property is based on the following lemma, an important auxiliary result about the instantiation of transparent variables. Intuitively it states that if we have a pattern t with type τ and we change its variables by other expressions, the only way to obtain the same type τ for the substituted pattern is by changing the transparent variables for expressions with the same type.

Lemma 1.
Assume $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$*, where* $var(t) \subseteq \{\overline{X_n}\}$*. If* $\mathcal{A} \vdash t[\overline{X_n/t_n}] : \tau$ *and* X_j *is a transparent variable of* t *wrt.* $\mathcal{A}$ *then* $\mathcal{A} \vdash t_j : \tau_j$*.*

This is not guaranteed with opaque variables, and that is why we forbid their use in expressions. For example if we consider the pattern $snd\ X$, from $\mathcal{A} \oplus \{X : bool\}$ we can derive the type $bool \to bool$. If we replace X by the pattern 0 we can still derive the same type $bool \to bool$ for $snd\ 0$, but clearly it is false that 0 has type $bool$.

Chapter 4

Type Inference of Expressions

The typing relation $\vdash^{\bullet}$ lacks some properties that prevent its usage as a type-checker mechanism in a compiler for a functional logic language. First, in spite of the syntax-directed style, the rules for $\vdash$ and $\vdash^{\bullet}$ have a bad operational behavior: at some steps they need to guess types. Second, the set of types related to an expression can be infinite due to polymorphism. Finally, the typing relation needs all the assumptions for the symbols in order to work. To overcome these problems, type systems usually are accompanied with a type inference algorithm which returns a valid type for an expression and also establishes the types for some symbols in the expression.

In this chapter we will present a type inference relation an algorithm that will provide an easy method for finding, given $\mathcal{A}$, a type for any expression. In fact, they go a step further: they will find a type and a substitution that are most general[1].

4.1 Type inference rules

In this section we will present a type inference procedure with a relational style, in order to show the similarities with the typing relation presented in Chapter 3. In the next section we will present the same procedure expressed in an algorithmic way, similar to algorithm $\mathscr{W}$ [48, 20].

The first style can be found in the relation $\Vdash$ of Figure 4.1. There is an inference rule for every derivation rule, so the type inference is also syntax-directed. Given a set of assumptions $\mathcal{A}$ and an

[1] in a sense that will be made precise later.

$$\textbf{[iID]}\quad \frac{}{\mathcal{A} \Vdash s : \tau | id} \quad \text{if} \quad s \in DC \cup FS \cup \mathcal{DV} \wedge \mathcal{A}(s) = \sigma \wedge \sigma \succ_{var} \tau$$

$$\textbf{[iAPP]}\quad \frac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash e_1 e_2 : \alpha\pi | \pi_1\pi_2\pi} \quad \text{if} \quad \alpha\ fresh\ type\ variable \wedge \pi = mgu(\tau_1\pi_2, \tau_2 \to \alpha)$$

$$\textbf{[i}\Lambda\textbf{]}\quad \frac{(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\}) \Vdash t : \tau_t | \pi_t \quad (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t \Vdash e : \tau | \pi}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi \to \tau | \pi_t\pi} \quad \text{if} \quad \{\overline{X_n}\} = var(t) \wedge \overline{\alpha_n}\ fresh\ type\ variables$$

$$\textbf{[iLET}_m\textbf{]}\quad \frac{(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\}) \Vdash t : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \textbf{let}_m\ t = e_1\ \textbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2} \quad \text{if} \quad \{\overline{X_n}\} = var(t) \wedge \overline{\alpha_n}\ fresh\ type\ variables \wedge \pi = mgu(\tau_t\pi_1, \tau_1)$$

$$\textbf{[iLET}^X_{pm}\textbf{]}\quad \frac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\} \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \textbf{let}_{pm}\ X = e_1\ \textbf{in}\ e_2 : \tau_2 | \pi_1\pi_2}$$

$$\textbf{[iLET}^h_{pm}\textbf{]}\quad \frac{(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\}) \Vdash h\ t_1 \ldots t_m : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \textbf{let}_{pm}\ h\ t_1 \ldots t_n = e_1\ \textbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2} \quad \text{if} \quad \{\overline{X_n}\} = var(h\ t_1 \ldots t_m) \wedge \overline{\alpha_n}\ fresh\ type\ variables \wedge \pi = mgu(\tau_t\pi_1, \tau_1)$$

$$\textbf{[iLET}_p\textbf{]}\quad \frac{(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\}) \Vdash t : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_t\pi_1\pi \oplus \{\overline{X_n : Gen(\alpha_n\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi)}\} \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \textbf{let}_p\ t = e_1\ \textbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2} \quad \text{if} \quad \{\overline{X_n}\} = var(t) \wedge \overline{\alpha_n}\ fresh\ type\ variables \wedge \pi = mgu(\tau_t\pi_1, \tau_1)$$

Figure 4.1: **Type inference rules**

$$\textbf{[iP]}\frac{\mathcal{A} \Vdash e : \tau | \pi}{\mathcal{A} \Vdash^{\bullet} e : \tau | \pi} \quad \text{if} \quad critVar_{\mathcal{A}\pi}(e) = \emptyset$$

Figure 4.2: **Extended type inference rules**

expression e, $\Vdash$ relates them with a type τ and a substitution π. Intuitively, τ is the "most general" type that can be given to e, and π is the "minimum" substitution we need to apply to $\mathcal{A}$ in order to be able to derive a type for e. This way, the inference will not only find a type for an expression but it also solves the constraints that $\mathcal{A}$ must fulfil by means of the substitution. This is better illustrated by an example. Suppose that we have a function f such that we do not know its exact type, only that takes an argument of some type and returns something of the same type; and we know also that $true$ has type $bool$. It will be expressed in a set of assumptions $\mathcal{A} \equiv \{f : \alpha \to \alpha, true : bool\}$. If we try to derive a type for $f\ true$ we could not, because α is different from $bool$. But if we infer the type for $f\ true$ the result will be $bool$ and the substitution $[\alpha \mapsto bool]$. This result expresses that in order to derive a type for $f\ true$ we first need to instantiate the type of f to $bool \to bool$. With these new assumptions, $bool$ will be the "most general type" for $f\ true$ (in this case $bool$ is the only type it can have).

The simplest rule is **[iID]**, which treats constructor/function symbols and variables. A valid type inferred for them is a variant of the type-scheme of the symbol in $\mathcal{A}$, and the substitution is the identity (id) because it is not needed to instantiate the assumptions to derive a type for s. **[iAPP]** infers types for expression application. It first infers a type τ_1 for e_1 from the original assumptions, obtaining also a substitution π_1. Then it infers a type τ_2 for e_2 but applying the substitution π_1 to the set of assumptions. Finally, it finds the most general unifier π of the type τ_1 inferred for e_1 and the functional type $\tau_2 \to \alpha$, being α a free variable. If they unify then τ_1 will be a functional type whose left part will unify with τ_2 and the right part with α. This way $\alpha\pi$ will be the type resulting of the application, and e_2 will have a type compatible with the left part of the type of e_2. Notice that the way the substitutions are used forces an order on how we must infer the types for the subexpressions. In this case we first infer the type for e_1, and then the obtained substitution is used to infer the type for e_2. We could also have chosen to first infer the type for e_2 and then for e_1, and finally find the most general unifier. The result will be the same, since the method only collects the constraints of every subexpression and finds a type and a substitution that are most general. In all the inference rules we have chosen to infer types for subexpressions from left to right, so this is the way the obtained substitutions are used (they appear up and down in the rules).

[iΛ] infers types for λ-expressions. It first extends $\mathcal{A}$ with fresh assumptions over the variables of the pattern, and infers a type τ_t for it. Then with the obtained assumptions for the variables it infers a type τ for the body of the λ-expressions. The final type inferred for the expression will be $\tau_t\pi \to \tau$, and the final substitution will be the composition $\pi_t\pi$.

The four rules for inferring types for let expressions are very similar, and only differ in the kind of polymorphism they provide to the variables of the pattern. As **[iΛ]** they first extends $\mathcal{A}$ with fresh assumptions over the variables of the pattern, and infers a type τ_t for it. Then (using the obtained assumptions from the previous step) it infers a type τ_1 for the right-hand side of the binding. At this

point a unification step is needed, in order to check that the obtained type $\tau_t\pi_1$ for the pattern and τ_1 for the expression in the right-hand side of the binding unify, i.e., they are compatible. Finally, using the previously found substitutions it infers a type τ_2 for e_2, which will be the type inferred for the whole expression. The final substitution will be the composition of the found substitutions in the order they are found. This last step differs between the four rules. **[iLET$_m$]** and **[iLET$_h^h$]** treat the variables of the pattern monomorphically, so the assumptions used in the inference for e_2 are just the obtained from the previous steps. On the other hand, rules **[iLET$_p^X$]** and **[iLET$_p$]** treat the variables of the pattern polymorphically. Instead of using the assumptions obtained from the previous steps in the final inference they generalize the assumptions over the variables in the pattern. This generalization is done wrt. the set of assumptions up to the moment, so the found substitutions are taking into account. Notice that in **[iLET$_p^X$]** the steps of inferring a type for pattern (a variable) and the subsequent unification do not appear in the rule. Inferring a type for the variable X using the extended set of assumptions $\mathcal{A} \oplus \{X : \alpha\}$ will yield to the type α and the substitution id. Then, the unifier of α and the type τ_1 obtained for e_1 will be trivially $\pi \equiv [\alpha/\tau_1]$. In this case $\alpha\pi$ will be the same as τ_1, so we have made the simplification of eliminate these two steps and generalize τ_1 directly in the last inference. Notice that in all cases the unification performs *occurs check*. The following example shows some type inferences for expressions:

Example 14 (Type inference for expressions)**.**

1. *This is the λ-abstraction associated to co, the problematic function in Example 3. Although it has critical variables we can infer the type of the λ-abstraction using $\Vdash$. Let $\mathcal{A}_1$ be the set of assumptions $\{snd : \forall\alpha, \beta.\alpha \to \beta \to \beta\}$*

$$\textbf{[i}\Lambda\textbf{]}\ \frac{\textbf{[iAPP]}\ \dfrac{\textbf{[iID]}\ \dfrac{}{\mathcal{A}_1 \oplus \{X : \gamma\} \Vdash snd : \gamma \to \delta \to \delta | id}\quad \textbf{[iID]}\ \dfrac{}{\mathcal{A}_1 \oplus \{X : \gamma\} \Vdash X : \gamma | id}}{\mathcal{A}_1 \oplus \{X : \gamma\} \Vdash snd\ X : \delta \to \delta | [\epsilon/\delta \to \delta]}\quad \textbf{[iID]}\ \dfrac{}{\mathcal{A}_1 \oplus \{X : \gamma\} \Vdash X : \gamma}}{\mathcal{A}_1 \Vdash \lambda snd\ X.X : (\delta \to \delta) \to \gamma | [\epsilon/\delta \to \delta]}$$

   In this case ϵ is a fresh variable generated to unify $\gamma \to \epsilon$ with $\gamma \to \delta \to \delta$.

2. *The following is an example of completely monomorphic let. Let $\mathcal{A}_2$ be the set of assumptions $\{id : \forall\alpha.\alpha \to \alpha, 0 : int\}$*

$$\textbf{[iLET}_m\textbf{]}\ \frac{\textbf{[iID]}\ \dfrac{}{\mathcal{A}_2 \oplus \{F : \beta\} \Vdash F : \beta | id}\quad \textbf{[iID]}\ \dfrac{}{\mathcal{A}_2 \Vdash id : \gamma \to \gamma | id}\quad \textbf{[iAPP]}\ \dfrac{\mathcal{A}_2 \oplus \{F : \gamma \to \gamma\} \Vdash F : \gamma \to \gamma | id \quad \mathcal{A}_2 \oplus \{F : \gamma \to \gamma\} \Vdash 0 : int | id}{\mathcal{A}_2 \oplus \{F : \gamma \to \gamma\} \Vdash F\ 0 : int | [\gamma/int, \delta/int]}}{\mathcal{A}_2 \Vdash let_m\ F = id\ in\ F\ 0 : int | [\beta/int \to int, \gamma/int, \delta/int]}$$

   In this example δ is a fresh variable generated to unify $\gamma \to \gamma$ and $int \to \delta$

The extended type inference relation $\Vdash^\bullet$ is based on $\Vdash$. It only contains one rule, **[iP]**, as can be seen in Figure 4.2. It simply finds a type and a substitution using $\Vdash$, and checks that the expression does not have critical variables, applying the substitution found to the set of assumptions.

It is usual in modern literature to split the inference process into two: the first step collects the constraints between the types involved in the expression and the second solves them. This kind of "Constraint-Based Typing" is explained in [58], where equality constraints are collected for the expressions and solved with an unification algorithm. Others authors as [68] or [7] have gone further in this topic, developing general frameworks where the constraint system is a paremeter. In this work we have decided to keep the solving mechanism in the rules because the algorithm for finding most general unifiers is well-known and is simple enough, and the whole inference process looks more compact.

4.2 Type inference algorithm

In the previous section we have presented inference as relations $\Vdash$ and $\Vdash^\bullet$, which relates $(\mathcal{A}, e)$ with (τ, π). But these relations can be viewed easily as recursive algorithms. Each rule (except **[iID]**) has the same relation with subexpressions in the premises. And these premises in the rules of Figure 4.1 have a clear order, due to the use of the obtained substitutions. The computation of variants in rule **[iID]** is easily achieved by an algorithm, since it is just changing the quantified variables by fresh ones. And there exist well-known algorithms for finding most general unifiers for syntactic unification, e.g. Robinson [63] or Martelli-Montanari [46, 9]. Therefore the inference can be viewed as an algorithm that given $(\mathcal{A}, e)$ returns (τ, π), where each rule of the original relation expresses a different case and that fails if none of the rules can be applied. This algorithm, called $\mathcal{E}$ after the famous algorithm $\mathscr{W}$ [48, 20, 19], appears in Figure 4.3.

The situation is the same with relation $\Vdash^\bullet$. The relation $\Vdash$ appears in its premises (a call to $\mathcal{E}$ algorithm), and it is also needed a checking of critical variables. The order between these two steps is clear: we need first to find the substitution, and then use it to check the critical variables. By the characterization of the opaque variables in Proposition 2 we can use $\Vdash$, i.e., algorithm $\mathcal{E}$, to find the opaque variables of a pattern. Then we can use the definition of critical variables (Definition 13) directly as a recursive algorithm. The whole algorithm for inferring, called $\mathcal{E}^\bullet$, can be found in Figure 4.2.

Notice that both algorithms $\mathcal{E}$ and $\mathcal{E}^\bullet$ are terminating. In the first case there are only recursive calls where the the expression to infer types decreases strictly. In the base case, when the expression is a symbol, the computation of a variant always finishes. The algorithm for finding the most general unifiers is known to terminate, so the whole algorithm $\mathcal{E}$ is terminating. In the case of algorithm $\mathcal{E}^\bullet$ it calls $\mathcal{E}$, which is terminating, and also checks for critical variables. This checking terminates because the recursive algorithm based on the definition is terminating (the argument decreases strictly), and the computation of opaque variables that it uses also terminates (because it uses $\mathcal{E}$). So the whole algorithm

$\mathcal{E}(s, \mathcal{A}) = (\tau, id)$ if
$(s : \sigma) \in \mathcal{A} \wedge \sigma \succ_{var} \tau$.

$\mathcal{E}(e_1\ e_2, \mathcal{A}) = (\alpha\pi, \pi_1\pi_2\pi)$ if
$\mathcal{E}(e_1, \mathcal{A}) = (\tau_1, \pi_1)$ and
$\mathcal{E}(e_2, \mathcal{A}\pi_1) = (\tau_2, \pi_2)$ and
$\pi = mgu(\tau_1\pi_2, \tau_2 \rightarrow \alpha)$
where α is fresh.

$\mathcal{E}(\lambda t.e, \mathcal{A}) = (\tau_t\pi \rightarrow \tau, \pi_t\pi)$ if
$\mathcal{E}(t, \mathcal{A} \oplus \{\overline{X_n : \alpha_n}\}) = (\tau_t, \pi_t)$ and
$\mathcal{E}(e, (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t) = (\tau, \pi)$
where $\overline{\alpha_n}$ are fresh and $\{\overline{X_n}\} = var(t)$.

$\mathcal{E}(let_m\ t = e_1\ in\ e_2, \mathcal{A}) = (\tau_2, \pi_t\pi_1\pi\pi_2)$ if
$\mathcal{E}(t, \mathcal{A} \oplus \{\overline{X_n : \alpha_n}\}) = (\tau_t, \pi_t)$ and
$\mathcal{E}(e_1, \mathcal{A}) = (\tau_1, \pi_1)$ and
$\pi = mgu(\tau_t\pi_1, \tau_1)$ and
$\mathcal{E}(e_2, (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi) = (\tau_2, \pi_2)$
where $\overline{\alpha_n}$ are fresh and $\{\overline{X_n}\} = var(t)$.

$\mathcal{E}(let_{pm}\ X = e_1\ in\ e_2, \mathcal{A}) = (\tau_2, \pi_1\pi_2)$ if
$\mathcal{E}(e_1, \mathcal{A}) = (\tau_1, \pi_1)$ and
$\mathcal{E}(e_2, \mathcal{A} \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\}) = (\tau_2, \pi_2)$.

$\mathcal{E}(let_{pm}\ h\ t_1 \ldots t_m = e_1\ in\ e_2, \mathcal{A}) = (\tau_2, \pi_t\pi_1\pi\pi_2)$ if
$\mathcal{E}(h\ t_1 \ldots t_m, \mathcal{A} \oplus \{\overline{X_n : \alpha_n}\}) = (\tau_t, \pi_t)$ and
$\mathcal{E}(e_1, \mathcal{A}) = (\tau_1, \pi_1)$ and
$\pi = mgu(\tau_t\pi_1, \tau_1)$ and
$\mathcal{E}(e_2, (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi) = (\tau_2, \pi_2)$
where $\overline{\alpha_n}$ are fresh and $\{\overline{X_n}\} = var(h\ t_1 \ldots t_m)$.

$\mathcal{E}(let_p\ t = e_1\ in\ e_2, \mathcal{A}) = (\tau_2, \pi_t\pi_1\pi\pi_2)$ if
$\mathcal{E}(t, \mathcal{A} \oplus \{\overline{X_n : \alpha_n}\}) = (\tau_t, \pi_t)$ and
$\mathcal{E}(e_1, \mathcal{A}) = (\tau_1, \pi_1)$ and
$\pi = mgu(\tau_t\pi_1, \tau_1)$ and
$\mathcal{E}(e_2, (\mathcal{A} \oplus \{\overline{X_n : Gen(\alpha_n\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi)}\})\pi_t\pi_1\pi) = (\tau_2, \pi_2)$
where $\overline{\alpha_n}$ are fresh and $\{\overline{X_n}\} = var(t)$.

NOTE: if none of the above conditions is met then $\mathcal{E}$ fails.

Figure 4.3: **Type inference as an algorithm $\mathcal{E}$**

$\mathcal{E}^\bullet(e, \mathcal{A}) = (\tau, \pi)$ if
$\mathcal{E}(e, \mathcal{A}) = (\tau, \pi)$ and
$critVar_{\mathcal{A}\pi}(e) = \emptyset$.

NOTE: if none of the conditions is met then $\mathcal{E}^\bullet$ fails.

Figure 4.4: **Extended type inference as an algorithm $\mathcal{E}^\bullet$**

$\mathcal{E}^\bullet$ is terminating as well.

4.3 Properties of the type inference

As we have said, inference is not only an effective way of finding some type for an expression. It has also two important features: it recollects the "minimum" constraints that the set of assumptions must fulfil in order to be able to derive any type for the expression (by means of the substitution) and it also finds the most general type for the expression. In this section we will formalize these results.

The first important property of the inference is that the type found is a valid, i.e., there exists a type derivation for that type using the set of assumptions affected by the substitution found. This result is called the *soundness* of the inference, and it is stated in the next theorem. Here we will use the symbol $|\!|\!\vdash^?$ for any of the inference relations: $|\!|\!\vdash$ or $|\!|\!\vdash^\bullet$.

Theorem 4 (Soundness of $|\!|\!\vdash^?$)**.**
$\mathcal{A} \; |\!|\!\vdash^? \; e : \tau | \pi \Longrightarrow \mathcal{A}\pi \vdash^? e : \tau$

The substitution π found by the inference has the important property of being more general than any other that permits to derive a type for the expression, i.e., they will be instances of π. The inferred type has a similar property: if applying a substitution to the set of assumptions it is possible to derive a type, the inferred type will be more general. The following theorem states this result for the inference relation $|\!|\!\vdash$.

Theorem 5 (Completeness of $|\!|\!\vdash$ wrt. $\vdash$)**.**
If $\mathcal{A}\pi' \vdash e : \tau'$ *then* $\exists \tau, \pi, \pi''. \; \mathcal{A} \; |\!|\!\vdash e : \tau | \pi \wedge \mathcal{A}\pi\pi'' = \mathcal{A}\pi' \wedge \tau\pi'' = \tau'$.

If a derivation exists then the inference will succeed, finding a substitution and a type that are the most general (upon renaming). Notice that this strong property is only valid with the relation $|\!|\!\vdash$. A result

similar to Theorem 5 cannot be obtained for $\Vdash^\bullet$ because of critical variables, as the following example shows.

Example 15 (Inexistence of a most general typing substitution)**.**
Let $\mathcal{A}$ be the set of assumptions $\{snd' : \alpha \to bool \to bool\}$ and consider the following two valid derivations

$$\mathcal{D}_1 : \quad \mathcal{A}[\alpha/bool] \equiv \{snd' : bool \to bool \to bool\} \vdash^\bullet \lambda(snd'\ X).X : (bool \to bool) \to bool$$

and

$$\mathcal{D}_2 : \quad \mathcal{A}[\alpha/int] \equiv \{snd' : int \to bool \to bool\} \vdash^\bullet \lambda(snd'\ X).X : (bool \to bool) \to int$$

It is clear that there is not a substitution more general than $[\alpha/bool]$ and $[\alpha/int]$ which makes possible a type derivation for $\lambda(snd'\ X).X$. The only substitution more general than these two will be $[\alpha/\beta]$ (for some β), converting X in an opaque variable in $snd'\ X$ and thereby a critical variable in the expression.

In spite of this fact, we will see that $\Vdash^\bullet$ is still able to find a most general substitution when it exists. To formalize that, we will need the notion of the set collecting all type substitution π such that $\mathcal{A}\pi$ gives some type to e.

Definition 16 (Typing substitutions of e)**.**
$\Pi^e_{\mathcal{A}} = \{\pi \in \mathcal{TS}ubst \mid \exists \tau \in SType.\ \mathcal{A}\pi \vdash e : \tau\}$
${}^\bullet\Pi^e_{\mathcal{A}} = \{\pi \in \mathcal{TS}ubst \mid \exists \tau \in SType.\ \mathcal{A}\pi \vdash^\bullet e : \tau\}$

Now we are ready to formulate our result regarding the maximality of $\Vdash^\bullet$.

Theorem 6 (Maximality of $\Vdash^\bullet$)**.**

a) *${}^\bullet\Pi^e_{\mathcal{A}}$ has a maximum element $\Longleftrightarrow \exists \tau_g \in SType, \pi_g \in \mathcal{TS}ubst.\mathcal{A} \Vdash^\bullet e : \tau_g|\pi_g$.*

b) *If $\mathcal{A}\pi' \vdash^\bullet e : \tau'$ and $\mathcal{A} \Vdash^\bullet e : \tau|\pi$ then exists a type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ and $\tau' = \tau\pi''$.*

The part $a)$ of Theorem 6 states that if ${}^\bullet\Pi^e_{\mathcal{A}}$ has a maximum element (wrt. the relation $\lesssim$ between substitutions) then the inference will succeed, and conversely, if the inference succeeds then the set ${}^\bullet\Pi^e_{\mathcal{A}}$ will have a maximum element. $b)$ states that if there exists a type derivation $\mathcal{A}\pi' \vdash^\bullet e : \tau'$ and the type inference succeeds, then the substitution and the type found by the inference will be more general than those appearing in the type derivation. Combining $a)$ and $b)$ we have that when there exists a most

general substitution that allows to derive a type for an expression, the type inference will succeed and will find it.

For finishing this chapter we will introduce a remark about the type inference that will be used in the proofs of Appendix A. This observation shows that the result of the type inference is unique upon renaming of fresh variables generated during the inference, so if they are replaced by other fresh variables the result remains valid.

Remark 5 (Uniqueness of the type inference)**.**
The result of a type inference is unique upon renaming of fresh type variables. In a type inference $\mathcal{A} \Vdash e : \tau|\pi$ *the variables in* $FTV(\tau)$, $Dom(\pi)$ *or* $Rng(\pi)$ *which do not occur in* $FTV(\mathcal{A})$ *are fresh variables generated by the inference process, so the result will remain valid if we change those variables by any other fresh types variables.*

Chapter 5

Type Inference of Programs

The type inference of a program is the method that, given a program (possibly with type declarations for some functions), finds the types for the functions in the program and checks if the provided type declarations are correct. In the previous chapter we presented type inference only for expressions. In the functional programming setting, type inference does not need to distinguish between programs and expressions, because the program can be incorporated in the expression by means of let expressions and λ-abstractions. This way, the results given for expressions are also valid for programs. But in our framework it is different, because our semantics (*let*-rewriting) does not support λ-abstractions and our let expressions do not define new functions but only perform pattern matching. Thereby in our case we cannot express a program as an expression, and we need to provide an explicit method for inferring the types of a whole program. This explicit method is very important, as it is the core of the type stage in a compiler.

In this chapter we will present two methods for inferring types for programs. *Block inference* (Section 5.2) infers types for a whole program "as is". *Stratified inference* (Section 5.3) relies on the previous one, but uses the dependencies between functions to split the program into blocks; which permits it to infer types for more programs and/or need less type declarations. We will also introduce *polymorphic recursion*, a situation which is not handled conveniently in the classical Damas & Milner type system.

5.1 Polymorphic recursion

Polymorphic recursion is the situation when different occurrences of the defined function appear with different instances of the type of the function in its own recursive definition. This situation also arise when dealing with a block of function definitions, which forces to handle them together. These situations are illustrated in the following examples:

Example 16 (Inherent polymorphic recursion)**.**

data tree A = empty | node A (tree (tree A))

flatmap :: (A → [B]) → [A] → [B]
flatmap F [] = []
flatmap F (X:Xs) = (F X) ++ (flatmap F Xs)

collect Tree = case Tree of
empty → []
node N T → N : (flatmap collect (collect T))

This example has been extracted from [37]. It comes from the ML mailing list, and has arisen in practice. Here tree A *is a polymorphic tree type, and* flatmap *is a mapping function that also concatenates the resulting lists. In this example* collect *is used with two types:* $tree\ (tree\ A) \rightarrow [tree\ A]$ *and* $tree\ A \rightarrow [A]$*. If polymorphic recursion is not allowed, this program will be rejected, since* collect *should only have one type in the definition. It can be easily proved in GHC or Hugs. If we supply the explicit type declaration for* collect, $\forall\alpha.tree\ \alpha \rightarrow [\alpha]$*, the program becomes typable.*

Example 17 (Block of definitions with polymorphic recursion)**.**

map F L → if (null L) then L else (F (head L)) : map F (tail L)
squarelist L → map ($\lambda X.X * X$*) L*
notlist L → map not L

This example has been extracted from [50], and it only presents polymorphic recursion when the three definitions are handled together. In this case, map *is used with two different types:* $(int \rightarrow int) \rightarrow [int] \rightarrow [int]$ *in* $squarelist$ *and* $(bool \rightarrow bool) \rightarrow [bool] \rightarrow [bool]$ *in* $notlist$*. This block of definitions will be correct if polymorphic recursion is supported, because both types are generic instances of the type-scheme for* map*:* $\forall\alpha,\beta.(\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$*. If polymorphic recursion is not supported this block will be rejected, since* map *can only have type* $(int \rightarrow int) \rightarrow [int] \rightarrow [int]$ *or* $(bool \rightarrow bool) \rightarrow [bool] \rightarrow [bool]$*, but not both.*

Some other interesting examples needing polymorphic recursion can be found in [26], where the authors compare their tipability with different type systems.

In the original Damas & Milner type system polymorphic recursion is not supported, because they provide a **FIX** operator which forces the defined function and all its occurrences in the recursive definition to have the same simple type. Mycroft [50] extends this type system by allowing polymorphic recursion, providing a **fix** operator which lets the defined function and its occurrences to have the same type-scheme. This way different occurrences could have different generic instances of them. [50] also extends the type inference to support polymorphic recursion, but the resulting algorithm is not terminating in all cases. Later, [32, 39] proved (independently) that type inference in the presence of polymorphic recursion is undecidable, due to it can be reduced to the problem of semiunification, which has been proved to be semidecidable[38]. Therefore, any algorithm that tries to infer types in the presence of polymorphic recursion can not terminate for certain cases. But if type-schemes are provided for the polymorphic recursive functions the problem becomes decidable, since it is inferring types for the other symbols and checking the provided type-schemes are correct.

5.2 Block inference

The type inference procedure for a program takes a set of assumptions $\mathcal{A}$ and a program $\mathcal{P}$ and returns a type substitution π. The set $\mathcal{A}$ must contain assumptions for all the symbols in the program, even for the functions defined in $\mathcal{P}$. We want to reflect the fact that in practice some defined functions may come with an explicit type declaration. Indeed this is a frequent way of documenting a program. Furthermore, type declarations are sometimes a real need, as we have explained in the previous section about polymorphic recursion. Therefore, for some of the functions –those for which we want to infer types– the assumption will be simply a fresh type variable, to be instantiated by the inference process. For the rest, the assumption will be a closed type-scheme, to be checked by the procedure. At the end, $\mathcal{A}\pi$ will contain the assumptions for all the defined functions in the program.

Type inference of a program is defined as follows:

Definition 17 (Type Inference of a Program)**.**
The procedure $\mathcal{B}$ for type inference of a program $\{rule_1, \ldots, rule_m\}$ is defined as:

$$\mathcal{B}(\mathcal{A}, \{rule_1, \ldots, rule_m\}) = \pi, \textit{if}$$

1. $\mathcal{A} \Vdash^{\bullet} (\varphi(rule_1), \ldots, \varphi(rule_m)) : (\tau_1, \ldots, \tau_m)|\pi$.

2. *Let $f^1 \dots f^k$ be the function symbols of the rules $rule_i$ in $\mathcal{P}$ such that $\mathcal{A}(f^i)$ is a closed type-scheme, and τ^i the type obtained for $rule_i$ in step 1. Then τ^i must be a variant of $\mathcal{A}(f^i)$.*

φ is a transformation from rules to expressions defined as:

$$\varphi(f\ t_1 \dots t_n \to e) = pair\ \lambda t_1 \dots . \lambda t_n . e\ f$$

*where **()** is the usual tuple constructor, with type **()** : $\forall\overline{\alpha_i}.\alpha_1 \to \dots \alpha_m \to (\alpha_1, \dots, \alpha_m)$; and **pair** is a special constructor of tuples of two elements of the same type, with type **pair** : $\forall\alpha.\alpha \to \alpha \to \alpha$.*

Procedure $\mathcal{B}$ first transforms the program into a single tuple expression, with as much elements as rules in the program. Each rule is transformed into a special "pair" of elements of the same type, by means of the transformation φ. The first element is the λ-abstraction associated to the rule, and the second is the function symbol defined in the rule. This special "pair" forces the λ-abstraction and the defined function symbol to have the same type if the inference succeed. The occurrence of the defined function symbol in each pair also forces all the rules of the same function to have the same inferred type. The absence of critical variables in the rules is guaranteed by the fact that the λ-abstractions appearing in the expression have not critical variables. The second step checks that every defined functions with a closed type-scheme in the assumptions (type declaration in the program), the inferred types for its rules are variants of that type-scheme. If we did not perform this check the program would not be well-typed wrt. $\mathcal{A}\pi$, so the procedure would not be sound.

Notice that types inferred for the functions are simple types. In order to obtain type-schemes we need and extra step of generalization, as discussed in Section 5.3.

5.2.1 Properties of block inference

The procedure $\mathcal{B}$ has two important properties. If $\mathcal{B}(\mathcal{A}, \mathcal{P})$ finds a substitution π, then the program $\mathcal{P}$ is well typed with respect to the assumptions $\mathcal{A}\pi$. This soundness property is stated in the following theorem.

Theorem 7 (Soundness of $\mathcal{B}$)**.**
If $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ then $wt_{\mathcal{A}\pi}(\mathcal{P})$.

Theorem 8 states the second property of $\mathcal{B}$, called maximality. It states that if the procedure $\mathcal{B}$ succeeds, it finds a substitution that is more general than any other which makes the program well-typed.

Theorem 8 (Maximality of $\mathcal{B}$)**.**
If $wt_{\mathcal{A}\pi'}(\mathcal{P})$ and $\mathcal{B}(\mathcal{A},\mathcal{P}) = \pi$ then $\exists\pi''$ such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$.

It is not true in general that the existence of a well-typing substitution π' implies the existence of a most general one. Here we show a counterexample of this fact, which is very similar to that showed for expressions in Example 15.

Example 18 (Inexistence of a most general well-typing substitution for a program)**.**
Let $\mathcal{P}$ be the program $\{snd\ X\ Y\ \rightarrow\ Y, co\ (snd\ X)\ \rightarrow\ X\}$, and let $\mathcal{A}$ be the set of assumptions $\{snd : \alpha \rightarrow bool \rightarrow bool, co : (bool \rightarrow bool) \rightarrow \alpha\}$. The substitution $\pi_1 \equiv [\alpha/bool]$ makes the program well-typed, since $\mathcal{A}\pi_1 \vdash^{\bullet} \lambda X.\lambda Y.Y : bool \rightarrow bool \rightarrow bool$, which is a variant of $\mathcal{A}\pi_1(snd) = bool \rightarrow bool \rightarrow bool$; and $\mathcal{A}\pi_1 \vdash^{\bullet} \lambda(snd\ X).X : (bool \rightarrow bool) \rightarrow bool$, which is a variant of $\mathcal{A}\pi_1(co) = (bool \rightarrow bool) \rightarrow bool$. The program is also well-typed with $\pi_2 \equiv [\alpha/int]$. In both cases the pattern $snd\ X$ is not problematic because the substitutions eliminate the type variables of the type of snd, so X is not opaque and therefore it is not critical.

The only substitution more general than π_1 and π_2 is $\pi \equiv [\alpha/\beta]$, being β any type variable. But this substitution ***does not make the program well-typed****, because $\mathcal{A}\pi \not\vdash^{\bullet} \lambda(snd\ X).X : (bool \rightarrow bool) \rightarrow \beta$. In this case X is opaque in $snd\ X$ (since $\mathcal{A}\pi \oplus \{X : \beta\} \vdash snd\ X : bool \rightarrow bool$ and $FTV(\beta) \nsubseteq FTV(bool \rightarrow bool)$), and therefore it is critical.*

5.3 Stratified inference

The procedure $\mathcal{B}$ handles programs has a block, without detecting the dependencies between functions. This has the disadvantage of usually needing much explicit type declarations in order to succeed, because of polymorphic recursion. It is known that splitting a program into blocks of mutually dependent functions and inferring the types in order may reduce the need of providing explicit type-schemes. This situation is illustrated in the following example.

Example 19 (Block inference vs. stratified inference)**.**
Assume the block of definitions from Example 17:

map F L → if (null L) then L else (F (head L)) : map F (tail L)
squarelist L → map (λX.X ∗ X) L
notlist L → map not L

An attempt to apply the procedure $\mathcal{B}$ to infer types will fail, because it is not possible for map to have types $(int \rightarrow int) \rightarrow [int] \rightarrow int$ and $(bool \rightarrow bool) \rightarrow [bool] \rightarrow bool$ at the same time. We

will need to provide explicitly the type-scheme for $map : \forall\alpha,\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$ *in order to the type inference to succeed, yielding types* $squarelist : [int] \to [int]$ *and* $notlist : [bool] \to [bool]$*. If we examine the block, we discover that both* $squarelist$ *and* $notlist$ *depend on* map*, but map does not depend on any other function in the block. Therefore we can split the block into three ordered blocks of mutually dependent functions, where each block depends only on the previous ones:*

$$\mathcal{P}_1 \equiv \{map\ F\ L \to if\ (null\ L)\ then\ L\ else\ (F\ (head\ L)) : map\ F\ (tail\ L)\}$$
$$\mathcal{P}_2 \equiv \{squarelist\ L \to map\ (\lambda X.X * X)\ L\}\ \textit{and}$$
$$\mathcal{P}_3 \equiv \{notlist\ L \to map\ not\ L\}$$

If we infer types for $\mathcal{P}_1$ *using* $\mathcal{B}$*, we obtain for* map *the type* $(\alpha \to \beta) \to [\alpha] \to [\beta]$*, that is completely generalized to* $\forall\alpha,\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$*. With this assumption we can continue inferring types for* $\mathcal{P}_2$ *and* $\mathcal{P}_3$ *and generalizing them, obtaining the expected types.*

Definition 18 (Stratified inference)**.**
A general stratified inference *procedure is defined in terms of the basic inference* $\mathcal{B}$ *in the following way:*

1. *Create the dependency graph of the program.*
2. *Calculate the graph of strongly connected components (SCCs) from the dependency graph.*
3. *Create blocks containing all the rules of the functions in the SCCs.*
4. *Sort the blocks according to any topological sort of the SCC graph.*
5. *For each block in the sorted list:*
   (a) *Infer types with* $\mathcal{B}$ *using the the obtained types from the previous blocks.*
   (b) *Generalize all the obtained types for the current block.*

The dependency graph will contain as much vertices as defined functions in the program. In this graph there will be an edge between f and g if g appears in the right-hand side or in any of the patterns of the left-hand side in any rule of f. The computation of the SCC graph from the dependency graph is easily achieved using any of the well-known algorithms that we will briefly discuss in next chapter. Each strongly connected component will contain mutually dependent function. Since they are dependent their types also depend on the other, so they must be inferred jointly. It is necessary to topologically sort the SCCs in order to assure that we have all the type information that will be used while inferring types for a block. Otherwise, we may need the type of a function that we have not inferred yet, or that has not been checked (if we have an explicit type declaration).

It is possible to simplify the generalization step 5b. We can assume that the original assumptions we use to infer types for a program contains always closed-types schemes from the Prelude or other files, so there will not be free type variables. After inferring types for each block using $\mathcal{B}$, we generalize *all* type variables appearing in them, without checking if the type variables occur free in the set of assumptions. If they appear free they will only be in an assumption for a defined function in the same block (because the rest are closed type-schemes), so the functions will be part of the same mutually dependent definition. In this case we can safely generalize them, obtaining a closed set of assumptions again.

Notice that although stratified inference needs less explicit type-schemes, programs involving inherent polymorphic recursion still require explicit type-schemes in order to infer their types, as in the program of Example 16.

Chapter 6

Implementation

In this chapter we will explain the most important issues about the implementation of the previously defined algorithms of inference for expression/programs and stratified inference in Prolog [21, 66]. We have chosen Prolog because this implementation will be part of the new version 3 of $\mathcal{TOY}$, which will be released soon. Since Prolog as been chosen as the main programming language in the development of $\mathcal{TOY}$ 3, concretely Sicstus Prolog 4 [5], this implementation will fit easily. The vast majority of the code follows the ISO Prolog Standard, making it easy to adapt from Sicstus 4 to any other Prolog system. Indeed, it has been proved in SWI Prolog [73, 6], a free and popular Prolog system.

The current $\mathcal{TOY}$ compiler [69] does not check the absence of opaque patterns in the program, so it accepts as valid programs like the polymorphic casting of Example 3. Apart from the detection of critical variables, which solves the previous problem, our implementation contributes another improvement over the existing type checking procedure in $\mathcal{TOY}$: it allows programs with inherent polymorphic recursion when suitable explicit types signatures are provided. Currently, the $\mathcal{TOY}$ compiler splits the program into blocks of mutually dependent functions, as our stratified inference procedure. Then, it discards the explicit type signatures provided by the programmer and infers types for the block. When it finishes, it compares the inferred types to the user-defined types: if the inferred type for a function is more general, it keeps the user-defined type (showing a warning message), otherwise it raises and error because the user-defined type is more general than the inferred one. This way of performing the type inference prevents the usage of explicit type signatures to allow polymorphic recursion, since the type inference is always performed without using the user-defined types. Therefore, programs like the one in Example 16 are rejected by the compiler, even when the explicit type for $collect$ is provided. In our implementation the program will be accepted when the explicit type $\forall\alpha.tree\ \alpha \to [\alpha]$ is provided, and it will be rejected

otherwise. Besides these improvements, our implementation also supports the three different kinds of local declarations explained in this work. The existing $\mathcal{TOY}$ compiler only permits local definitions by means of `where` declarations, which are treated in a completely monomorphic way[1]. Therefore, our implementation provides more flexibility to the programmer in local declarations.

The code is divided in 4 files. `operators.pl` contains the definition of the operators used in the syntax of expressions and types. `infer.pl` contains the predicates for inferring types of expressions and programs. `stronglycc.pl` contains the implementation of Tarjan's algorithm [70, 74] for finding strongly connected components in graphs. Finally, `stratified.pl` contains the predicates for inferring types for programs in the stratified way, using inference for expressions and programs and the algorithm for the strongly connected component.

The complete code can be found in:
`http://gpd.sip.ucm.es/enrique/systems/stratifiedInference/stratifiedInference.zip`.

6.1 Syntax of the terms. Operators.

In the implementation, expressions, programs and types are represented as Prolog terms. In order to make the syntax more readable, we have defined some infix operators. Data variables are represented as Prolog variables, and constructor/function symbols as Prolog atoms. Expression application is represented as `Expr @ Expr`, using the left associative infix operator `@`. λ-abstractions are terms constructed using the functor `lambda/2`, where the first argument is the pattern and the second the expression. Let expressions are represented using functors `letm/2`, `letpm/2` and `letp/2`; where the first argument is the pattern, the second the expression to bind and the third the expression of the body. Programs rules are constructed with the functor `rule/3` where the first argument is the function (an atom), the second is the list of patterns of its left-hand side, and the third is the expression of the body. Finally, a program is a Prolog list of program rules. The syntax of types in Prolog is summarized in the next figure.

Types need also be represented in the implementation (Figure 6.2). Type variables, like data variables, are represented by Prolog variables. This is very convenient since unification and occurs check is handled automatically by the Prolog system, and no explicit handling of substitutions is required. Type constructors are represented by common Prolog terms. For functional types we use the declared infix operator −», which is right associative. Type application uses the same operator `@` as expression application. Type-schemes are represented using the functor `tscheme/2`, where the first argument is a list of

[1] since λ-lifting is used to eliminate `where` declarations.

Data variable	↦	`Prolog variable`
Constructor/function symbol	↦	`Prolog atom`
Expression application	↦	`Expr @ Expr`
λ-abstractions	↦	`lambda(Pat,Expr)`
Let expressions	↦	`letm(Pat,Expr,Expr)`
		`|letpm(Pat,Expr,Expr)`
		`|letp(Pat,Expr,Expr)`
Program rule	↦	`rule(Func,ListOfPatterns,Body)`
Program	↦	`Prolog list of program rules`

Figure 6.1: **Syntax of expressions and programs in Prolog**

variables (those that are quantified) and the second is a simple type. An assumption is constructed using the operator `::`. This operator is infix and not associative. The left hand side is a variable or an atom (data variable or constructor/function symbol) and the right-hand side is a simple type. Finally, sets of assumptions are represented as lists of assumptions. In this implementation we also use facts of the form `type(Identifier,SimpleType)` to store assumptions for constructors/function symbols due to efficiency and simplicity. To infer types we need the types of all the symbols, in particular the types of the built-in symbols, because they can appear elsewhere. If we only use a list of assumptions, every time we need to find a variant of the type of a built-in symbol during the inference, we have to search in the whole list, which is time consuming. But built-in symbols have always closed type-schemes, so they can be stored easily in `type/2` facts because free variables will not be a problem. This way, searching for them is performed efficiently performed due to the indexing mechanism of the Prolog system. Moreover, calculating a variant of a type stored in a `type/2` fact is simpler and more efficient than a type stored in a list of assumptions, as we will see in next section. Therefore, the list of assumptions will only contain assumptions for variables and the symbols in the program, and the rest will be stored in `type/2` facts.

Type variable	↦	`Prolog variable`
Type constructor	↦	`Prolog atom`
Functional type	↦	`SimpleType1 -» SimpleType2`
Type application	↦	`SimpleType1 @ SimpleType2`
Type-scheme	↦	`tscheme(VariableList, SimpleType)`
Assumption	↦	`identifier ::  TypeScheme`
Set of assumptions	↦	`Prolog list of assumptions`
Assumption fact	↦	`type(Identifier,SimpleType)`

Figure 6.2: **Syntax of types in Prolog**

6.2 Inference algorithms: `infer.pl`

This file contains the implementation of the inference algorithms $\mathcal{E}$ and $\mathcal{E}^\bullet$ for expressions and $\mathcal{B}$ for programs. The most interesting predicates for expressions are:

- `infer( +Assump, +Expr, -SimpleType )`
  It returns the inferred type for `Expr` using the set of assumptions `Assump`. The implementation follows directly the pseudocode of Figure 4.3. In this case it is not necessary to handle substitutions explicitly, because the Prolog system treats them automatically. We have included *ad-hoc* rules to handle float and built-in types.

- `inferp( +Assump, +Expr, -Type )`
  It implements the algorithm presented in Figure 4.4 using `infer/3` and `critVar/2`.

- `critVar( +Assump, +Expr )` This predicate checks that `Expr` has not any critical variable wrt. `Assump`. It implements the alternative characterization of opaque variable of Proposition 2, so it uses the `infer/3` predicate.

- `variant( +TScheme, -SimpleType )`
  This predicate takes a type-scheme `tscheme( Gen, Type )` as an argument and returns a simple type that is a variant. To create a variant we create a substitution (explicitly represented as a list of pairs) from the generalized variables in `Gen` to fresh variables, and apply it to `Type`. It is important this substitution by fresh variables, because if not different variants will share variables as if they were free.

  This predicate is only used with type-schemes stored in the list of assumptions. If the assumption is stored in a fact, a call to `type(Id,Type)` will directly return a variant, since the Prolog system will change the variables by fresh ones.

- `gen( +Assump, +Type, -TypeScheme )`
  Returns the type-scheme resulting of the generalization of `Type` wrt. the set of assumptions `Assump`. It looks in the free type variables of `Assump` (using `ftv/2`), and includes in the list of generalized variables of `TypeScheme` those variables of `Type` not appearing in it.

- `fv( +Expr, -EVars )`
  It returns a list of the free variables of the expression.

- `ftv( +TypeScheme, -LVars )` / `ftv( +Assump, -LVars )`
  It returns a list of the free variables of the type-scheme.

For programs, the most interesting predicates are:

- `b( ?Assump, +Prog )`
  Given a set of assumptions, infers the type of the function symbols defined in a program using algorithm $\mathcal{B}$ (Definition 17). Notice that `Assump` is an input/output argument, so if `b/2` succeeds the assumptions for the symbols will have been instantiated to the proper types. If some symbols in `Assump` contain a closed type-scheme as assumption, it will check that the inferred type is a variant.

- `b( +Assump, +Prog, +CheckList)`
  Previous predicate `b/2` is defined in term of this. It act inferring types rule by rule. If it finds a rule defining a function whose symbol has a type-scheme as an assumption, it adds the inferred type to `CheckList`. When it finishes inferring type for the rules, processes the list to check that the inferred types are variants of the type-schemes.

- `check_variant( +Type, +Func, +Assump )`
  Checks that `Type` is a variant of the type-scheme stored in `Assump` for the function `Func`.

6.3 Strongly connected components algorithm: `stronglycc.pl`

This module implements Tarjan's algorithm [70] for computing strongly connected components (SCCs) of a graph. There exists more algorithms for computing SCCs, as Kosaraju's [18] or Gabow's [22]. A complete explanation of the three algorithms and their code in C++ can be found in [64]. Kosaraju's its the classical algorithm for computing SCCs, its easy to implement and runs in $O(|V| + |E|)$ time when the graph is represented *adjacency lists*, which is optimal because any algorithm for computing SCCs must examine all the vertices (V) and edges (E). Kosaraju's algorithm is based in the fact that the SCCs of a graph and the transposed graph (the same vertices but changing the direction of the edges) are the same. Although it is previous, Tarjan's algorithm can be viewed as an improvement of Kosaraju's. It runs also in $O(|V| + |E|)$ time, but in practice it is more efficient than Kosaraju's because it performs one depth-first search on the graph instead of two. It is based on the fact that the SCCs form the subtrees of the search tree, the roots of which are the roots of the SCCs. Gabow's algorithm is the less known algorithm of the three, and it also performs one depth-first search, but in practice its efficiency is comparable with Tarjan's, and its code is more complex. We have chosen Tarjan's algorithm because of its balance between efficiency and simplicity of code. The implementation of the algorithm has been based on the pseudocode appearing in the Wikipedia [74], that we reproduce in Figure 6.3.

```
1  Input: Graph G = (V, E)
2
3  index = 0                               // DFS node number counter
4  S = empty                               // An empty stack of nodes
5  forall v in V do
6    if (v.index is undefined)             // Start a DFS at each node
7      tarjan(v)                           // we haven't visited yet
8
9  procedure tarjan(v)
10   v.index = index                       // Set the depth index for v
11   v.lowlink = index
12   index = index + 1
13   S.push(v)                             // Push v on the stack
14   forall (v, v') in E do                // Consider successors of v
15     if (v'.index is undefined)  // Was successor v' visited?
16         tarjan(v')                      // Recurse
17         v.lowlink = min(v.lowlink, v'.lowlink)
18     elif (v' is in S)             // Was successor v' in stack S?
19         v.lowlink = min(v.lowlink, v'.index)
20   if (v.lowlink == v.index)             // Is v the root of an SCC?
21     print "SCC:"
22     repeat
23       v' = S.pop
24       print v'
25     until (v' == v)
```

Figure 6.3: **Pseudocode of Tarjan's algorithm - Wikipedia** [74]

We store the information of the edges of the graph using Prolog facts `edge(A,B)`, which means that there exists an edge between A and B (arrow from A to B). We also use a list containing the vertices of the graph. This list is important because Tarjan's algorithm only finds the SCCs of the vertices that are reachable from the original vertex, so we need the list to call it from every unvisited vertex. Furthermore, some vertices may not be involved in any edge, so they will not appear in any `edge/2` fact. These vertices will form a SCC by themselves. As they appear in the vertex list, they will be eventually used as initial vertices for the algorithm, so the associated SCC will be handled correctly. We also need to store persistent information about the vertices as their index (a number indicating the order of discovery) and the lowest index of the vertices reachable from them. For efficiency reasons, we have chosen to store also whether a node has been pushed in the stack instead of searching in the list. All this persisten information is stored using dynamic predicates `index(Node, Index)`, `lowlink(Node, LowLink)` and `instack(Node)` that will be asserted and retracted during run-time.

The most important predicates of the file are:

- `scc(+VertexList)`
  The main predicate. It calculates the strongly connected components of the graph represented in the `edge/2` facts, and whose vertices appear in the list `VertexList`. The result is asserted as a fact `scc_list(SCC_List)` containing the SCCs, each of them represented as a list of vertices.
- `tarjan( +Vertex, +Index, +Stack, -NIndex, -NStack )`
  It calculates the SCCs of the graph starting from the vertex `Vertex` using Tarjan's algorithm. `Index` is the index counter, and `Stack` is the actual stack of vertices. It returns `NIndex`, the new index counter; and `NStack`, the new stack of vertices. It uses the predicate `findall/3` to find the list of all the successor vertices of the current vertex.
- `visit_successors( +Vertex, +SuccessorList, +Index, +Stack, -NewIndex, -NewStack )`
  It visits all the vertices in `SuccessorList` which are the successors of the vertex `Vertex`. This predicate implements the loop of the Tarjan's algorithm that appears in lines 14-19 of Figure 6.3.

6.4 Stratified inference: `stratified.pl`

This file contains the implementation of the stratified inference for programs. This inference method splits the whole program into sets of mutually dependent functions and infers the types set by set gener-

alizing the obtained types, as explained in Section 5.3. The main predicate is `stratified/1`, whose argument is the path of the file containing the program to infer types. The file contains a simple Prolog program compound by `type/2` and `rule/3` facts. The syntax of this file is a simplification of a intermediate representation of a source program containing only the relevant information for type inference. `type/2` facts act as type signatures in usual functional languages, and `rule/3` facts represent function rules as explained in Section 6.1.

The main predicates in this file are:

- `stratified( +FileName )`
  It infers types for the program stored in `FileName` using the stratified method. It builds the dependency graph, calculates the SCCs and infers type SCC by SCC. The inferred types are stored in `type/2` facts.

- `retrieve_dependencies`
  It captures the dependencies between functions from the rules and stores them using `edge/2` facts, i.e., it builds the dependency graph from the `rule/3` facts.

- `inferSCC( +SCC )`
  It infers types for a SCC, storing the generalization of the inferred types using `type/2` facts.

- `genAssump( +Assump )`
  It generalizes the types appearing in `Assump`, adding them to the database using `type/2` facts. All the variables are generalized, even if they occur in `Assump`. `Assump` is assumed to contain only $\{FunctionSymbol :: SimpleType\}$ assumptions. This predicate is used by `inferSCC/1` after inferring the types for a SCC using algorithm $\mathcal{B}$.

- `initialAssump( +SCC, -Assump )`
  It generates the initial assumptions for the functions in the list `SCC`. This predicate is used before inferring types for a SCC. If the function `F` in `Assump` appears in a `type(F,Type)` fact, no initial assumption is created. Otherwise, an assumption `F::A` (being `A` a new type variable) will appear in `Assump`.

Chapter 7

Conclusions and Future Work

In this work we have presented a type system for functional logic languages that solves a well-known problem which appears when HO patterns are handled naively in the type system. The results of this work have been accepted as a paper in the 18^{th} International Workshop on Functional and (Constraint) Logic Programming (WFLP'09) held in Brasilia on June 28th. The paper has been also accepted to appear in the Lecture Notes in Computer Science volume associated to the workshop.

7.1 Contributions of this work

- We have developed a type system based on Damas & Milner that solves the problems that appears due to opacity with HO patterns (Section 3.1.2). This type system is more flexible than the proposed in [24], since it supports some HO patterns that the type system in [24] rejects. The key resides in the notion of *critical variables*. In [24] they forbid any opaque pattern, whether it contains opaque variables or not. In our type system we permit opaque patterns whenever they do not contain critical variables, i.e., opaque variables which appear in the rest of the expression. Therefore, a pattern like $snd\ X$ which is rejected in [24] is allowed in our type system if the variable X does not occur in the rest of the expression. We also allow opaque patterns without opaque variables like $snd\ true$, which is rejected in [24] as well.

  The uniform treatment of the opaque and critical variables allows us to be more liberal with the types of data constructors, dropping the traditional restriction of transparency required to them. Therefore data constructors can have non transparent types, and patterns containing them will be accepted or rejected depending on the presence of critical variables, as in HO patterns.

- We have clarified the different kinds of polymorphism that bound variables can have in let expressions (Section 3.1.1). This is an issue that varies greatly between implementations of functional and functional logic languages, and that is usually not documented nor formalized. We provide a type system that supports three different kinds of polymorphism for let expressions explicitly, without relying in any program transformation.

- The extension of the syntax of expressions in [24] with λ-abstractions and let expressions is other of our contributions (Section 3.1.1). We have explicitly addressed the occurrence of compound patterns in the arguments of λ-abstractions and in the bindings of let expressions, a fact usually obviated in the literature or avoided by means of transformations. We provide a type system which explicitly supports compound patterns.

- The type system developed has the subject reduction property (Section 3.3): expressions keep their types after evaluation steps. This is a key property of type system, since it assures the absence of type errors during run-time. For example, consider the boolean expression $e_1 \wedge true$ in which e_1 has type $bool$. We can be sure that after evaluation steps in e_1 the resulting expression will have type $bool$, so eventually we will be able to apply the rule for $\wedge$[1]. This also assures that a well-type expression will not lose its type during evaluation. In [24] they prove this property wrt. the Goal-Oriented Rewriting Calculus (GORC). In this work we have proved subject reduction wrt. *let*-rewriting [44], a small-step operational semantics for functional logic languages closer to real computations.

- We have provided an inference algorithm for finding the type of expressions (Chapter 4). It is stronger than a simple type-checker, since it also reconstructs the types of the symbols in the expression when necessary. In the practice, this allows the programmer to omit the majority of the type declarations in the program, as in languages like ML or Haskell. This algorithm is sound (the types it finds are correct wrt. the type system) and "maximal" (it finds the most general type when it exists). It cannot be complete because some expressions which admit type do not have a most general one. In these cases the algorithm fails.

- We have also provided two explicit methods for inferring types for programs. In the functional setting this is usually omitted because programs can be understood as chains of let expressions defining functions accompanied by an expression to evaluate. In our framework this is not the case, since our let expressions cannot define functions but perform pattern matching. These two algorithms rely on the algorithm for inferring types for expressions. The block inference in Section 5.2 treats the program as a block. It has been proved to be sound (the program is well-typed with the found types) and "maximal" (if it succeed it finds the most general types for the program). It is

[1] if e_1 is eventually evaluated to a pattern

not always possible to find the most general types because, as in the case of expressions, they do not always exists. The stratified inference in Section 5.3 acts more cleverly, splitting the program into mutually dependent components. This reduces the need of explicit type declarations.

- We have implemented the stratified inference in Prolog. It is an improvement over the existing type inference procedure in the $\mathcal{TOY}$ compiler [69] because it detects critical variables and allows polymorphic recursion when explicit types signatures are provided. It also supports the three different kinds of local declarations (let_m, let_{pm} and let_p), in contrast to the monomorphic $where$ declarations of the existing $\mathcal{TOY}$ system. Our implementation currently supports programs in a very simplified format, but it will be soon extended and integrated in the $\mathcal{TOY}$ compiler as the type stage. The source code of the stratified inference can be found in `http://gpd.sip.ucm.es/enrique/systems/stratifiedInference/stratifiedInference.zip`.

- We have paid special attention to the formal aspects of the work. Therefore we have developed detailed proofs of the properties of the type system, the subject reduction property, and the soundness and completeness of the inference algorithms wrt. the type system. All the proofs can be found in Appendix A.

7.2 Future work

In spite of the obtained results, there is still much work to do. Our first task is to integrate the stratified inference into the $\mathcal{TOY}$ compiler. This will solve the problems that currently appear in the presence of opaque HO patterns. To do that, it will be very useful the stratified inference for simplified programs that we have already implemented in Prolog. Type errors management is a crucial issue in programming systems, and it is not addressed in the current implementation. We plan to extend the implementation with a robust type error management in order to produce informative errors to $\mathcal{TOY}$ users.

Other of our aims is to generalize the subject reduction property to *let*-narrowing [44], in order to support the entire expressiveness of functional logic programming. *Let*-rewriting is a semantics with *call-time choice* and nondeterministic functions, but it does not support the binding of variables during execution, which is needed by programs as in Example 2. The generalization of the subject reduction property to *let*-narrowing presents well-known problems [24, 8] like the instantation of higher order variables, which is illustrated in the following example:

Example 20 (Binding of higher order variables)**.**
Let $\mathcal{P}$ be the program $\{add\ zero\ X\ \rightarrow\ X, add\ (succ\ X)\ Y\ \rightarrow\ succ(add\ X\ Y)\}$. If we have the

expression $F\ X == true$, let-*narrowing could find the substitution* $[F/add\ zero, X/true]$. *This is a solution, since* $\mathcal{P} \vdash add\ z\ true \rightarrow^{l} true$, *but clearly* $add\ z\ true$ *is ill-typed.* Let-*narrowing finds this substitution because it does not consider type information but only the program rules, so it cannot perform any type checking while guessing instances. Therefore,* let-*narrowing needs to be extended to support type information for programs and expressions.*

We also want to handle extra variables (variables occurring only in the right-hand side of the rules), a problem not investigated from the point of view of types. These variables may not have a type fixed by the context where they appear, and it is not clear how much degree of polymorphism give to them in such cases.

Our type system does not forbid opacity completely. It allows opaque variables when they do not appear in the rest of the expressions, i.e., when they are not critical. This notion of opacity in expressions is somehow similar to what happens with *existential types* [49] or *generalized algebraic data types* [16, 61, 57], a connection that we plan to further investigate in the future. Finally, we are interested in other known extensions of type system, like type classes [72, 52, 51] or generic programming [34, 35].

Bibliography

[1] Erlang programming language, official website. http://www.erlang.org.

[2] F# at microsoft research. http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/default.aspx.

[3] The glasgow haskell compiler. http://www.haskell.org/ghc/.

[4] The glorious glasgow haskell compilation system user's guide, version 6.8.3; chapter 8: Ghc language features; section 8.16. control over monomorphism. http://haskell.org/ghc/docs/6.8.3/html/users_guide/monomorphism.html.

[5] Sicstus prolog homepage. http://www.sics.se/isl/sicstuswww/site/index.html.

[6] Swi-prolog's home. http://www.swi-prolog.org.

[7] ALVES, S., AND FLORIDO, M. Type inference using constraint handling rules. *Electronic Notes Theoretical Compututer Science 64* (2002).

[8] ANTOY, S., AND TOLMACH, A. P. Typed higher-order narrowing without higher-order strategies. In *Fuji International Symposium on Functional and Logic Programming* (1999), pp. 335–353.

[9] BAADER, F., AND NIPKOW, T. *Term Rewriting and All That*. Cambridge University Press, 1998.

[10] BRASSEL, B. Post to the curry mailing list. http://www.informatik.uni-kiel.de/~curry/listarchive/0706.html, May 2008.

[11] BRUS, T. H., VAN EEKELEN, C. J. D., VAN LEER, M. O., AND PLASMEIJER, M. J. Clean: A language for functional graph rewriting. In *Proc. of a conference on Functional programming languages and computer architecture* (London, UK, 1987), Springer-Verlag, pp. 364–384.

[12] BUENO, F., CABEZA, D., CARRO, M., HERMENEGILDO, M., LÓPEZ, P., AND PUEBLA, G. *The Ciao Prolog System: A Next Generation Multi-Paradigm Programming Environment. REFERENCE MANUAL*, April 2006. Available at `http://clip.dia.fi.upm.es/Software/Ciao/ciao.pdf`.

[13] CARDELLI, L. Type systems. In *CRC Handbook of Computer Science and Engineering Handbook*, 2nd ed. CRC Press, 2004, ch. 97.

[14] CARDELLI, L., AND WEGNER, P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys 17* (1985), 471–522.

[15] CESARINI, F., AND THOMPSON, S. *Erlang Programming. A Concurrent Approach to Software Development*. O'Reilly, 2009.

[16] CHENEY, J., AND HINZE, R. First-class phantom types. Tech. rep., Cornell University, July 2003.

[17] CLEMENT, D. The natural dynamic semantics of mini-standard ml. In *II and Colloquium on Functional and Logic Programming and Specifications (CFLP) on TAPSOFT '87: Advanced Seminar on Foundations of Innovative Software Development* (New York, NY, USA, 1987), Springer-Verlag New York, Inc., pp. 67–81.

[18] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, second ed. The MIT Press, 2001.

[19] DAMAS, L. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, April 1985. Also appeared as Technical report CST-33-85.

[20] DAMAS, L., AND MILNER, R. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1982), ACM, pp. 207–212.

[21] DERANSART, P., ED-DBALI, A., AND CERVONI, L. *Prolog: The Standard. Reference Manual*. Springer, 1996.

[22] GABOW, H. N. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett. 74*, 3-4 (2000), 107–114.

[23] GONZÁLEZ-MORENO, J., HORTALÁ-GONZÁLEZ, M., AND RODRÍGUEZ-ARTALEJO, M. A higher order rewriting logic for functional logic programming. In *Proc. International Conference on Logic Programming (ICLP'97)* (1997), MIT Press, pp. 153–167.

[24] GONZÁLEZ-MORENO, J., HORTALÁ-GONZÁLEZ, T., AND RODRÍGUEZ-ARTALEJO, M. Polymorphic types in functional logic programming. In *Journal of Functional and Logic Programming* (2001), vol. 2001/S01, pp. 1–71. Special issue of selected papers contributed to the International Symposium on Functional and Logic Programming (FLOPS'99).

[25] GONZÁLEZ-MORENO, J. C., HORTALÁ-GONZÁLEZ, T., LÓPEZ-FRAGUAS, F., AND RODRÍGUEZ-ARTALEJO, M. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming 40*, 1 (1999), 47–87.

[26] HALLETT, J. J., AND KFOURY, A. J. Programming examples needing polymorphic recursion. Tech. rep., Boston University, March 2004.

[27] HANUS, M. Functional logic programming. `http://www.informatik.uni-kiel.de/~mh/FLP/`.

[28] HANUS, M. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming 19&20* (1994), 583–628.

[29] HANUS, M. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)* (2007), Springer LNCS 4670, pp. 45–75.

[30] HANUS (ED.), M. Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.

[31] HARPER, R., MACQUEEN, D., AND MILNER, R. Standard ml. Lfcs report series ecs-lfcs-86-2, University of Edinburgh, 1986.

[32] HENGLEIN, F. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst. 15*, 2 (1993), 253–289.

[33] HINDLEY, R. The principle type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc. 146* (1969), 29–60.

[34] HINZE, R. A new approach to generic functional programming. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2000), ACM, pp. 119–132.

[35] HINZE, R. Generics for the masses. *J. Funct. Program. 16*, 4-5 (2006), 451–483.

[36] HUSSMANN, H. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.

[37] JIM, T. What are principal typings and what are they good for? In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1996), ACM, pp. 42–53.

[38] KFOURY, A. J., TIURYN, J., AND URZYCZYN, P. The undecidability of the semi-unification problem. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing* (New York, NY, USA, 1990), ACM, pp. 468–476.

[39] KFOURY, A. J., TIURYN, J., AND URZYCZYN, P. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst. 15*, 2 (1993), 290–311.

[40] KFOURY, A. J., AND WELLS, J. B. Adding polymorphic abstraction to ml (detailed abstract), 1994.

[41] LEIJEN, D. HMF: Simple type inference for first-class polymorphism. In *13th ACM symp. of the International Conference on Functional Programming (ICFP'08)* (Sept. 2008). Extended version available as Microsoft Research technical report MSR-TR-2007-118, Sep 2007.

[42] LEIJEN, D. Flexible types: robust type inference for first-class polymorphism. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2009), ACM, pp. 66–77. Extended version available as Microsoft Research technical report MSR-TR-2008-55, Mar 2008.

[43] LLOYD, J. W. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming 3* (1999).

[44] LÓPEZ-FRAGUAS, F., RODRÍGUEZ-HORTALÁ, J., AND SÁNCHEZ-HERNÁNDEZ, J. Rewriting and call-time choice: the HO case. In *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS'08)* (2008), vol. 4989 of *LNCS*, Springer, pp. 147–162.

[45] LÓPEZ-FRAGUAS, F., AND SÁNCHEZ-HERNÁNDEZ, J. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)* (1999), Springer LNCS 1631, pp. 244–247.

[46] MARTELLI, A., AND MONTANARI, U. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4*, 2 (1982), 258–282.

[47] MCKINNA, J. Why dependent types matter. *SIGPLAN Not. 41*, 1 (2006), 1–1.

[48] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978), 348–375.

[49] MITCHELL, J. C., AND PLOTKIN, G. D. Abstract types have existential type. *ACM Trans. Program. Lang. Syst. 10*, 3 (1988), 470–502.

[50] MYCROFT, A. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming* (London, UK, 1984), Springer-Verlag, pp. 217–228.

[51] NIPKOW, T., AND PREHOFER, C. Type reconstruction for type classes. *Journal of Functional Programming 5*, 2 (1995), 201–224.

[52] PETERSON, J., AND JONES, M. Implementing type classes. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation* (New York, NY, USA, 1993), ACM, pp. 227–236.

[53] PEYTON JONES, S. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[54] PEYTON JONES, S., Ed. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[55] PEYTON JONES, S. Monomorphic pattern bindings - haskell prime wiki. `http://hackage.haskell.org/trac/haskell-prime/wiki/MonomorphicPatternBindings`, July 2006.

[56] PEYTON JONES, S., VYTINIOTIS, D., WEIRICH, S., AND SHIELDS, M. Practical type inference for arbitrary-rank types. *J. Funct. Program. 17*, 1 (2007), 1–82.

[57] PEYTON JONES, S., VYTINIOTIS, D., WEIRICH, S., AND WASHBURN, G. Simple unification-based type inference for gadts. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2006), ACM, pp. 50–61.

[58] PIERCE, B. P. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[59] PIERCE, B. P. *Advanced topics in types and programming languages*. MIT Press, Cambridge, MA, USA, 2005.

[60] PLASMEIJER, R., AND VAN EEKELEN, M. *Clean version 2.1 Language Report*, November 2002. Available at `http://clean.cs.ru.nl/contents/contents.html`.

[61] POTTIER, F., AND RÉGIS-GIANAS, Y. Stratified type inference for generalized algebraic data types. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2006), ACM, pp. 232–244.

[62] READE, C. *Elements of Functional Programming*. Addison-Wesley, 1989.

[63] ROBINSON, J. A. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 1 (1965), 23–41.

[64] SEDGEWICK, R. *Algorithms in C++, Part 5: Graph Algorithms*. Addison-Wesley Professional, 2002, ch. 19.8. Strong Components in Digraphs, pp. 205–216.

[65] SOMOGYI, Z., HENDERSON, F. J., AND CONWAY, T. C. The execution algorithm of mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming 29* (1996).

[66] STERLING, L., AND SHAPIRO, E. *The Art of Prolog*, second ed. The MIT Press, 1994.

[67] STRACHEY, C. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation 13*, 1-2 (2000), 11–49.

[68] SULZMANN, M. F. *A general framework for hindley/milner type systems with constraints*. PhD thesis, New Haven, CT, USA, 2000. Director-Hudak,, Paul.

[69] SÁNCHEZ, P. A., LEIVA, A. J. F., LUEZAS, A. G., FRAGUAS, F. J. L., ARTALEJO, M. R., AND PÉREZ, F. S. *TOY, A Multiparadigm Declarative Language. Version 2.3.1*, 2007. Available at http://toy.sourceforge.net/.

[70] TARJAN, R. Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1*, 2 (1972), 146–160.

[71] TOFTE, M., AND TALPIN, J.-P. Region-based memory management. *Inf. Comput. 132*, 2 (1997), 109–176.

[72] WADLER, P., AND BLOTT, S. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1989), ACM, pp. 60–76.

[73] WIELEMAKER, J. An overview of the SWI-Prolog programming environment. In *Proceedings of the 13th International Workshop on Logic Programming Environments* (Heverlee, Belgium, December 2003), F. Mesnard and A. Serebenik, Eds., Katholieke Universiteit Leuven, pp. 1–16. CW 371.

[74] WIKIPEDIA. Tarjan's strongly connected components algorithm. `http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm`.

Appendix A

Proofs

A.1 Previous remarks and easy facts not formally proved

In this section we will present some remarks about the type system and the programs that will be used in the proofs in the next section.

Remark 6.
If $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$ *then we can assume that* $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash e : \tau' | \pi$ *such that* $\mathcal{A}\pi = \mathcal{A}$.

Explanation. Intuitively, the inference finds a type which is more general than all the possible types for an expression, and also a type substitution which is necessary applying to the set of assumptions in order to derive a type for the expression. In this case it is possible from the original set of assumptions $\mathcal{A}$ to derive a type, so we do not need to change $\mathcal{A}$. Therefore the type substitution π from the inference would not need to affect $\mathcal{A}$, just only $\overline{\alpha_n}$ and the fresh variables generated during inference.

By Theorem 5 we know that there exists a type substitution π'' such that $\mathcal{A}\pi\pi'' = \mathcal{A}$ and $\tau'\pi'' = \tau$. This means that $\mathcal{A}\pi$ is just a renaming of some free type variables of $\mathcal{A}$, which are restored with the type substitution π''. Being $\mathcal{A}\pi$ a renaming of $\mathcal{A}$ is a consequence of the *mgu* algorithm used. In this case, during inference there will be some unifying steps between a free type variable α from $\mathcal{A}$ and a fresh one β. Clearly, both $[\alpha/\beta]$ and $[\beta/\alpha]$ are more general unifiers. In this cases if we choose the first, we will compute a substitution which will make $\mathcal{A}\pi$ a renaming of $\mathcal{A}$; but if we choose always to substitute the fresh type variables the set of assumption $\mathcal{A}\pi$ will remain the same as $\mathcal{A}$.

Remark 7.

In a type derivation $\mathcal{A} \vdash e : \tau$ will appear a type derivation for every subexpression e' of e. That is, the derivation will have a part of the tree rooted by $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e' : \tau'$, being τ' a suitable type for e', and being $\{\overline{X_n : \tau_n}\}$ a set of assumptions over variables of the expression e which have been introduced by the rules $[\Lambda]$, $[LET_m]$, $[LET^X_{pm}]$, $[LET^h_{pm}]$ or $[LET_p]$.
If the expression is a pattern, the set of assumptions $\{\overline{X_n : \tau_n}\}$ will be empty because the only rules used to type a pattern are $[ID]$ and $[APP]$.

Remark 8.
If $wt_{\mathcal{A}}(\mathcal{P})$ and $\mathcal{A}'$ is a set of assumptions for variables, then $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.
The reason is that $\mathcal{A}'$ does not change the assumptions for the function and constructor symbols in $\mathcal{A}$. Since there are not extra variables in the right hand sides, for every function rule in $\mathcal{P}$ the typing rule for the lambda expression will add assumptions for all the variables, shadowing the provided ones.

A.2 Proofs of lemmas and theorems

In this section we will present the proofs of the lemmas and theorems in the work. We will also include some auxiliary lemmas needed in the proofs of the main results.

The next lemma states that if we have a pattern t with type τ and we change its variables by other expressions, the only way to obtain the same type τ for the substituted pattern is by changing the transparent variables for expressions with the same type.

Lemma 1
Assume $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$, where $var(t) \subseteq \{\overline{X_n}\}$. If $\mathcal{A} \vdash t[\overline{X_n/t_n}] : \tau$ and X_j is a transparent variable of t wrt. $\mathcal{A}$ then $\mathcal{A} \vdash t_j : \tau_j$.

Proof. According to Observation 7, in the derivation of $\mathcal{A} \vdash t[\overline{X_n/t_n}] : \tau$ appear derivations for every subpattern t_i, and they have the form $\mathcal{A} \vdash t_i : \tau'_i$ for some τ'_i. We will prove that if X_j is a particular transparent variable of t, then $\tau_j = \tau'_j$. It is easy to see that taking the types $\overline{\tau'_n}$ as assumptions for the original variables $\overline{X_n}$ we can construct a derivation of $\mathcal{A} \oplus \{\overline{X_n : \tau'_n}\} \vdash t : \tau$, simply replacing the derivations for the subpatterns $\mathcal{A} \vdash t_i : \tau'_i$ with derivations for the variables $\mathcal{A} \oplus \{\overline{X_n : \tau'_n}\} \vdash X_i : \tau'_i$ in the original derivation for $\mathcal{A} \vdash t[\overline{X_n/s_n}] : \tau$. Since X_j is a transparent variable of t wrt $\mathcal{A}$, by definition $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_g | \pi_g$ and $FTV(\alpha_j \pi_g) \subseteq FTV(\tau_g)$. By Theorem 5, if any type for t can be derived from $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\}\pi_s$ then π_g must be more general than π_s. We know that there are (at least)

two substitutions π^1 and π^2 which can type t: $\pi^1 \equiv \{\overline{\alpha_n \mapsto \tau_n}\}$ and $\pi^2 \equiv \{\overline{\alpha_n \mapsto \tau'_n}\}$, so they must be more specific than π_g (i.e. there exist π, π' such that $\pi^1 = \pi_g\pi$ and $\pi^2 = \pi_g\pi'$. We also know (by Theorem 4) that $\mathcal{A} \oplus \{X_n : \alpha_n\} \Vdash t : \tau_g|\pi_g$ implies $(\mathcal{A} \oplus \{X_n : \alpha_n\})\pi_g \vdash t : \tau_g$, and by Theorem 1-a this implies that $(\mathcal{A} \oplus \{X_n : \alpha_n\})\pi_g\pi \vdash t : \tau_g\pi$; so $\tau_g\pi = \tau$ (the same thing happens with π': $\tau_g\pi' = \tau$).

At this point we can distinguish two cases:

A) X_j is transparent because of $FTV(\alpha_j\pi_g) = \emptyset$. Then $\tau_j = (\alpha_j\pi_g)\pi = \alpha_j\pi_g = (\alpha_j\pi_g)\pi' = \tau'_j$, because if $\alpha_j\pi_g$ does not have any free variable, it cannot be affected by any substitution.

B) X_j is transparent because of $FTV(\alpha_j\pi_g) \subseteq FTV(\tau_g)$. As $\tau_g\pi = \tau$ and $\tau_g\pi' = \tau$, then for every type variable β in $FTV(\tau_g)$ then $\beta\pi = \beta\pi'$. As every type variable β in $FTV(\alpha_j\pi_g)$ is also in $FTV(\tau_g)$ then as $\tau_j = (\alpha_j\pi_g)\pi = (\alpha_j\pi_g)\pi' = \tau'_j$.

□

The following lemma states that if the type inference of the expression e succeeds with $\mathcal{A}\pi$, then the type inference for the same expression will succeed with $\mathcal{A}$. Besides, the types and substitutions found will be related.

Lemma 2.
If $\mathcal{A}\pi \Vdash e : \tau_1|\pi_1$ *then* $\exists\tau_2 \in SType, \pi_2\ \pi'' \in \mathcal{TS}ubst\ s.t.\ \mathcal{A} \Vdash e : \tau_2|\pi_2$ *and* $\tau_2\pi'' = \tau_1$ *and* $\mathcal{A}\pi_2\pi'' = \mathcal{A}\pi\pi_1$.

Proof. By Theorem 4 $\mathcal{A}(\pi\pi_1) \Vdash e : \tau_1$. Then applying Theorem 5 $\mathcal{A} \Vdash e : \tau_2|\pi_2$ and there exists a type substitution $\pi'' \in \mathcal{TS}ubst$ such that $\tau_2\pi'' = \tau_1$ and $\mathcal{A}\pi_2\pi'' = \mathcal{A}\pi\pi_1$. □

The following lemma states that if we have an expression e which can have a type τ_x, all the generic instances of the type-scheme resulting of the generalization of that type are also valid types for the expression e.

Lemma 3.
If τ *is a generic instance of* $Gen(\tau_x, \mathcal{A})$ *and* $\mathcal{A} \vdash e : \tau_x$ *then* $\mathcal{A} \vdash e : \tau$.

Proof. Let be $Gen(\tau_x, \mathcal{A})$ of the form $\forall\overline{\alpha_i}.\tau_x$ according to the definition of generalization (Definition 11), being $\overline{\alpha_i}$ the type variables of τ_x which do not appear in $\mathcal{A}$. By definition of generic instance

(Definition 5) τ will have been constructed applying a type substitution $\pi \equiv [\overline{\alpha_i/\tau_i}]$ to τ_x from the variables $\overline{\alpha_i}$ to types, i.e., $\tau \equiv \tau_x\pi$. Because $\mathcal{A} \vdash e : \tau_x$ and the type system is closed under substitutions (by Theorem 1-a) we can build a type derivation $\mathcal{A}\pi \vdash e : \tau_x\pi$. But $\overline{\alpha_i}$ do not appear in $\mathcal{A}$ so $\mathcal{A}\pi \equiv \mathcal{A}$ and then $\mathcal{A} \vdash e : \tau$. □

The lemma that follows states that the definition of opaque variables in Definition 12 and the more operational characterization in Proposition 2 are equivalent.

Lemma 4 (Equivalence of the two characterizations of opaque variable)**.**
Let t be a pattern that admits type wrt. a given set of assumptions $\mathcal{A}$. Then

$$\exists\overline{\tau_n}, \tau \; s.t. \; \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau \; and \; FTV(\tau_i) \nsubseteq FTV(\tau)$$
$$\iff$$
$$\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_g|\pi_g \; and \; FTV(\alpha_i\pi_g) \nsubseteq FTV(\tau_g)$$

Proof.

$\Longrightarrow$ The type derivation can be written as $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})[\overline{\alpha_n/\tau_n}] \vdash t : \tau$, so by Theorem 5 $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_g|\pi_g$ and there exists some $\pi'' \in \mathcal{TS}ubst$ s.t. $\tau_g\pi'' = \tau$, $\mathcal{A}\pi_g\pi'' = \mathcal{A}$ and $\alpha_i\pi_g\pi'' = \tau_i$. We only need to prove that

$$FTV(\tau_i) \nsubseteq FTV(\tau) \Longrightarrow FTV(\alpha_i\pi_g) \nsubseteq FTV(\tau_g)$$

It is equivalent to prove

$$FTV(\alpha_i\pi_g) \subseteq FTV(\tau_g) \Longrightarrow FTV(\tau_i) \subseteq FTV(\tau)$$

which is trivial since $\alpha_i\pi_g\pi'' = \tau_i$ and $\tau_g\pi'' = \tau$, so

$$FTV(\alpha_i\pi_g) \subseteq FTV(\tau_g) \Longrightarrow FTV(\alpha_i\pi_g\pi'') \subseteq FTV(\tau_g\pi'')$$

$\Longleftarrow$ By Theorem 4 $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_g \vdash t : \tau_g$, and $FTV(\alpha_i\pi_g) \nsubseteq FTV(\tau_g)$. Since t admits type by Observation 6 $\mathcal{A}\pi_g = \mathcal{A}$, so $\mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_g}\} \vdash t : \tau_g$.

□

The following is an important result that allow us to prove that type derivations $\vdash^\bullet$ are closed under type substitutions. It states that if a pattern t has type with $\mathcal{A}$ and $\mathcal{A}\pi$ (for some assumptions for its

variables) then the opaque variables wrt. $\mathcal{A}\pi$ are a subset of the opaque variables wrt. $\mathcal{A}$, i.e., they decrease applying substitutions.

Lemma 5 (Decrease of opaque variables)**.**
If $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$ *and* $\mathcal{A}\pi \oplus \{\overline{X_n : \tau'_n}\} \vdash t : \tau'$ *then* $opaqueVar_{\mathcal{A}\pi}(t) \subseteq opaqueVar_{\mathcal{A}}(t)$.

Proof. Since $opaqueVar_{\mathcal{A}}(t) = var(t) \setminus transpVar_{\mathcal{A}}(e)$, then $opaqueVar_{\mathcal{A}\pi}(t) \subseteq opaqueVar_{\mathcal{A}}(t)$ is the same as $transpVar_{\mathcal{A}}(t) \subseteq transpVar_{\mathcal{A}\pi}(t)$. Then we have to prove that if a variable X_i of t is transparent wrt. $\mathcal{A}$ then it is also transparent wrt. $\mathcal{A}\pi$.

$\mathcal{A}\oplus\{\overline{X_n : \tau_n}\}$ is the same as $\mathcal{A}\oplus\{\overline{X_n : \alpha_n}\}[\overline{\alpha_n/\tau_n}]$, so by Theorem 5 we have that $\mathcal{A}\oplus\{\overline{X_n : \alpha_n}\} \Vdash t : \tau_1|\pi_1$. Then the transparent variables of t will be those X_i such that $FTV(\alpha_i\pi_1) \subseteq FTV(\tau_1)$.

$\mathcal{A}\pi \oplus \{\overline{X_n : \tau'_n}\}$ is the same as $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi[\overline{\alpha_n/\tau'_n}]$, because we can assume that the variables $\overline{\alpha_n}$ does not appear in π. Then by Theorem 5 $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi \Vdash t : \tau_2|\pi_2$, and by Lemma 2 there exists a type substitution π'' such that $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi\pi_2 = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_1\pi''$ and $\tau_2 = \tau_1\pi''$.

Therefore every data variable X_i which is transparent wrt. $\mathcal{A}$ will be also transparent wrt. $\mathcal{A}\pi$, because:

$$\begin{array}{cc} FTV(\alpha_i\pi_1) \subseteq FTV(\tau_1) & X_i \text{ is transparent wrt. } \mathcal{A} \\ FTV(\alpha_i\pi_1\pi'') \subseteq FTV(\tau_1\pi'') & \text{adding } \pi'' \text{ to both sides} \\ FTV(\alpha_i\pi\pi_2) \subseteq FTV(\tau_2) & X_i \text{ is transparent wrt. } \mathcal{A}\pi \end{array}$$

□

The next lemma is important when proving the inductive case of the subject reduction property. It states that in a type derivation you can replace one subexpression e by other expression e' if they have the same type in that place.

Lemma 6.
If $\mathcal{A} \vdash \mathcal{C}[e] : \tau$ *and in that derivation appear a derivation of the form* $\mathcal{A}\oplus\mathcal{A}' \vdash e : \tau'$*, and* $\mathcal{A}\oplus\mathcal{A}' \vdash e' : \tau'$ *then* $\mathcal{A} \vdash \mathcal{C}[e'] : \tau$.

Proof. We proceed by induction over the structure of the contexts:

[ ]) This case is straightforward because $[]e = e$ and $[]e' = e'$.

$\mathbf{e}_1\ \mathcal{C}$) Since $(e_1\ \mathcal{C})[e] = e_1\ \mathcal{C}[e]$, if we have a derivation for $\mathcal{A} \vdash (e_1\ \mathcal{C})[e]$ it must be of the form:

$$\textbf{[APP]}\ \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \qquad \mathcal{A} \vdash \mathcal{C}[e] : \tau_1}{\mathcal{A} \vdash e_1\ \mathcal{C}[e] : \tau}$$

A derivation of $\mathcal{A} \oplus \mathcal{A}' \vdash e : \tau'$ must appear in the whole derivation, so it must appear in the derivation $\mathcal{A} \vdash \mathcal{C}[e] : \tau_1$ (according to Observation 7). Since $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau'$ then by the Induction Hypothesis we can state that $\mathcal{A} \vdash \mathcal{C}[e'] : \tau_1$, and we can construct a derivation for $\mathcal{A} \vdash (e_1\ \mathcal{C})[e']$:

$$\textbf{[APP]}\ \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \qquad \mathcal{A} \vdash \mathcal{C}[e'] : \tau_1}{\mathcal{A} \vdash e_1\ \mathcal{C}[e'] : \tau}$$

$\mathcal{C}\ \mathbf{e}_1$) Similar to the previous case.

$\mathbf{let}_m$ X = $\mathcal{C}$ in $\mathbf{e}_1$) $(let_m\ X\ =\ \mathcal{C}\ in\ e_1)[e]$ is equal to $let_m\ X\ =\ \mathcal{C}[e]\ in\ e_1$, so a derivation of $\mathcal{A} \vdash (let_m\ X = \mathcal{C}\ in\ e_1)[e] : \tau$ must have the form:

$$\textbf{[LET}_m\textbf{]}\ \frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \qquad \mathcal{A} \vdash \mathcal{C}[e] : \tau_t \qquad \mathcal{A} \oplus \{X : \tau_t\} \vdash e_1 : \tau}{\mathcal{A} \vdash let_m\ X = \mathcal{C}[e]\ in\ e_1 : \tau}$$

Clearly, a derivation for $\mathcal{A} \oplus \mathcal{A}' \vdash e : \tau'$ will appear in the derivation for $\mathcal{A} \vdash \mathcal{C}[e] : \tau_t$ (Observation 7). Since $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau'$ then by the Induction Hypothesis we can state that $\mathcal{A} \vdash \mathcal{C}[e'] : \tau_t$. With this information we can construct a derivation for (let$_m$ X = $\mathcal{C}$ in e_1)[e']:

$$\textbf{[LET}_m\textbf{]}\ \frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \qquad \mathcal{A} \vdash \mathcal{C}[e'] : \tau_t \qquad \mathcal{A} \oplus \{X : \tau_t\} \vdash e_1 : \tau}{\mathcal{A} \vdash let_m\ X = \mathcal{C}[e']\ in\ e_1 : \tau}$$

$\mathbf{let}_m$ X = e_1 in $\mathcal{C}$) A type derivation of (let$_m$ X = e_1 in $\mathcal{C}$)[e] will have the form:

$$\textbf{[LET}_m\textbf{]}\ \frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \qquad \mathcal{A} \vdash e_1 : \tau_t \qquad \mathcal{A} \oplus \{X : \tau_t\} \vdash \mathcal{C}[e] : \tau}{\mathcal{A} \vdash let_m\ X = e_1\ in\ \mathcal{C}[e] : \tau}$$

By Observation 7, the derivation $\mathcal{A} \oplus \{X : \tau_t\} \vdash \mathcal{C}[e] : \tau$ will contain a derivation $(\mathcal{A} \oplus \{X : \tau_t\}) \oplus \mathcal{A}'' \vdash e : \tau'$. It is a premise that $(\mathcal{A} \oplus \{X : \tau_t\}) \oplus \mathcal{A}'' \vdash e' : \tau'$ (in this case $\mathcal{A}' = \{X : \tau_t\} \oplus \mathcal{A}''$), so by the Induction Hypothesis $\mathcal{A} \oplus \{X : \tau_t\} \vdash \mathcal{C}[e'] : \tau$ and we can construct a derivation $\mathcal{A} \vdash let_m\ X = e_1\ in\ \mathcal{C}[e'] : \tau$

$$\textbf{[LET}_m\textbf{]}\ \frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \qquad \mathcal{A} \vdash e_1 : \tau_t \qquad \mathcal{A} \oplus \{X : \tau_t\} \vdash \mathcal{C}[e'] : \tau}{\mathcal{A} \vdash let_m\ X = e_1\ in\ \mathcal{C}[e'] : \tau}$$

rest) The proofs for the cases $let_{pm}\ X\ =\ \mathcal{C}\ in\ e_1$, $let_{pm}\ X\ =\ e_1\ in\ \mathcal{C}$, $let_p\ X\ =\ \mathcal{C}\ in\ e_1$ and $let_p\ X = e_1\ in\ \mathcal{C}$ are similar to the proofs for let_m.

□

The lemma that follows states that in an expression e without critical variable, if you replace a variable by any other expression which has not critical variables, the resulting expression has not critical variables either.

Lemma 7.
If $critVar_{\mathcal{A}}(e) = \emptyset$ *and* $critVar_{\mathcal{A}}(e') = \emptyset$ *then* $critVar_{\mathcal{A}}(e[X/e']) = \emptyset$.

Proof. We will proceed by induction over the structure of e.

Base Case

c) Straightforward because $c[X/e'] = c$, so $critVar_{\mathcal{A}}(c[X/e']) = critVar_{\mathcal{A}}(c) = \emptyset$ from the premises.

f) The same as c.

X) In this case $X[X/e'] = e'$, and from the premises we know that $critVar_{\mathcal{A}}(e') = \emptyset$.

Y) Y is a variable distinct from X. Then $Y[X/e'] = Y$, and from the premises $critVar_{\mathcal{A}}(Y) = \emptyset$.

Induction Step

$e_1\ e_2$) By definition $critVar_{\mathcal{A}}(e_1\ e_2) = \emptyset$ if $critVar_{\mathcal{A}}(e_1) = \emptyset$ and $critVar_{\mathcal{A}}(e_2) = \emptyset$. Then by the Induction Hypothesis $critVar_{\mathcal{A}}(e_1[X/e']) = \emptyset$ and $critVar_{\mathcal{A}}(e_2[X/e']) = \emptyset$. By definition $(e_1\ e_2)[X/e'] = e_1[X/e']\ e_2[X/e']$, so:

$$\begin{aligned} critVar_{\mathcal{A}}((e_1\ e_2)[X/e']) &= critVar_{\mathcal{A}}(e_1[X/e']\ e_2[X/e']) \\ &= critVar_{\mathcal{A}}(e_1[X/e']) \cup critVar_{\mathcal{A}}(e_2[X/e']) \\ &= \emptyset \cup \emptyset \\ &= \emptyset \end{aligned}$$

$\lambda t.e$) We assume that $X \notin var(t)$ and $var(t) \cap FV(e') = \emptyset$. Since $critVar_{\mathcal{A}}(\lambda t.e) = \emptyset$ then $opaqueVar_{\mathcal{A}}(t) \cap FV(e) = \emptyset$ and $critVar_{\mathcal{A}}(e) = \emptyset$. $opaqueVar_{\mathcal{A}}(t) \subseteq var(t)$, so $opaqueVar_{\mathcal{A}}(t) \cap$

$FV(e') = \emptyset$. $opaqueVar_{\mathcal{A}}(t) \cap (FV(e) \cup FV(e')) = (opaqueVar_{\mathcal{A}}(t) \cap FV(e)) \cup (opaqueVar_{\mathcal{A}}(t) \cap FV(e')) = \emptyset$. Since $FV(e[X/e']) \subseteq FV(e) \cup FV(e')$, then $opaqueVar_{\mathcal{A}}(t) \cap FV(e[X/e']) = \emptyset$.

On the other hand by the Induction Hypothesis $critVar_{\mathcal{A}}(e[X/e']) = \emptyset$. Therefore

$$\begin{aligned} critVar_{\mathcal{A}}((\lambda t.e)[X/e']) &= critVar_{\mathcal{A}}(\lambda t.(e[X/e'])) \\ &= (opaqueVar_{\mathcal{A}}(t) \cap FV(e[X/e'])) \cup critVar_{\mathcal{A}}(e[X/e']) \\ &= \emptyset \cup \emptyset \\ &= \emptyset \end{aligned}$$

$let_m\ t = e_1\ in\ e_2$) We assume that $X \notin var(t)$, $var(t) \cap FV(e') = \emptyset$, and $var(t) \cap FV(e_1) = \emptyset$. Since $critVar_{\mathcal{A}}(let_m\ t = e_1\ in\ e_2) = \emptyset$ then $opaqueVar_{\mathcal{A}}(t) \cap FV(e_2) = \emptyset$, $critVar_{\mathcal{A}}(e_1) = \emptyset$ and $critVar_{\mathcal{A}}(e_2) = \emptyset$. From $var(t) \cap FV(e') = \emptyset$ and $opaqueVar_{\mathcal{A}}(t) \subseteq var(t)$ we know that $opaqueVar_{\mathcal{A}}(t) \cap FV(e') = \emptyset$. As in the previous case, $opaqueVar_{\mathcal{A}}(t) \cap (FV(e_2) \cup FV(e')) = \emptyset$ and $FV(e_2[X/e']) \subseteq FV(e_2) \cup FV(e')$, so $opaqueVar_{\mathcal{A}}(t) \cap FV(e_2[X/e']) = \emptyset$.

On the other hand by the Induction Hypothesis $critVar_{\mathcal{A}}(e_1[X/e']) = \emptyset$ and $critVar_{\mathcal{A}}(e_2[X/e']) = \emptyset$. Therefore

$$\begin{aligned} critVar_{\mathcal{A}}((let_m\ t = e_1\ in\ e_2)[X/e']) &= critVar_{\mathcal{A}}(let_m\ t = e_1[X/e']\ in\ e_2[X/e']) \\ &= (opaqueVar_{\mathcal{A}}(t) \cap FV(e_2[X/e'])) \cup \\ &\quad critVar_{\mathcal{A}}(e_1[X/e'])\ \cup\ critVar_{\mathcal{A}}(e_2[X/e']) \\ &= \emptyset \cup \emptyset \cup \emptyset \\ &= \emptyset \end{aligned}$$

The proofs for the let_{pm} and let_p cases are equal to the let_m case. □

The next lemma states that if all fresh variables (wrt. a set of assumptions $\mathcal{A}$) of a simple type τ do not appear in a type substitution π, then generalizing τ wrt. $\mathcal{A}$ and applying π is the same as generalizing the type with the substitution applied ($\tau\pi$) wrt. the set of assumptions with the substitution applied ($\mathcal{A}\pi$).

Lemma 8.
Let $\mathcal{A}$ be a set of assumptions, τ a type and $\pi \in \mathcal{TS}ubst$ such that for every type variable α which appears in τ and does not appear in $FTV(\mathcal{A})$ then $\alpha \notin Dom(\pi)$ and $\alpha \notin Rng(\pi)$. Then $(Gen(\tau, \mathcal{A}))\pi = Gen(\tau\pi, \mathcal{A}\pi)$.

Proof. We will study what happens with a type variable α of τ in both cases (types that are not variables are not modified by the generalization step).

- $\alpha \in FTV(\tau)$ and $\alpha \in FTV(\mathcal{A})$. In this case it cannot be generalized in $Gen(\tau, \mathcal{A})$, so in $(Gen(\tau, \mathcal{A}))\pi$ it will be transformed into $\alpha\pi$. Because $\alpha \in FTV(\mathcal{A})$, then all the variables in $\alpha\pi$ are in $FTV(\mathcal{A}\pi)$ and they cannot be generalized. Therefore in $Gen(\tau\pi, \mathcal{A}\pi)$ α will also be transformed into $\alpha\pi$.

- $\alpha \in FTV(\tau)$ and $\alpha \notin FTV(\mathcal{A})$. In this case α will be generalized in $Gen(\tau, \mathcal{A})$, and as π does not affect a generalized variable, it will remain in $(Gen(\tau, \mathcal{A}))\pi$. Because α is not in $Dom(\pi)$, then $\alpha\pi = \alpha$. $\alpha \notin Rng(\pi)$ and $\alpha \notin FTV(\mathcal{A})$, so it cannot appear in $\mathcal{A}\pi$. Therefore α will also be generalized in $Gen(\tau\pi, \mathcal{A}\pi)$.

□

The following lemma states that the type that results of generalizing a simple type τ wrt. a set of assumptions $\mathcal{A}$ and then applying a substitution π is always more general than the type that results of generalizing $\tau\pi$ wrt. $\mathcal{A}\pi$.

Lemma 9 (Generalization and substitutions)**.**
$Gen(\tau, \mathcal{A})\pi \succ Gen(\tau\pi, \mathcal{A}\pi)$

Proof. It is clear that if a type variable α in τ is not generalized in $Gen(\tau, \mathcal{A})$ (because it occurs in $FTV(\mathcal{A})$), then in the first type-scheme it will appear as $\alpha\pi$. In the second type scheme it will also appear as $\alpha\pi$ because all the variables in $\alpha\pi$ will be in $\mathcal{A}\pi$ (as $\alpha \in FTV(\mathcal{A}.)$). Therefore in every generic instance of the two type-schemes this part will be the same. On the other hand, if a type variable α is generalized in $Gen(\tau, \mathcal{A})$ then it will also appear generalized in $Gen(\tau, \mathcal{A})\pi$ (π will not affect it). It does not matter what happens with this part $\alpha\pi$ in $Gen(\tau\pi, \mathcal{A}\pi)$ because in every generic instance of $Gen(\tau, \mathcal{A})\pi$ the generalized α will be able to adopt all the types of any generic instance of the part $\alpha\pi$ in $Gen(\tau\pi, \mathcal{A}\pi)$. □

The lemma that follows shows an interesting property of the inference relation $\Vdash$. It states that if $\Vdash^\bullet$ succeeds with an expression e and a set of assumptions $\mathcal{A}$ then the sets of typing substitution $\Pi^e_{\mathcal{A}}$ and ${}^\bullet\Pi^e_{\mathcal{A}}$ are the same. In other words, all type substitution that gives a type to e wrt. $\vdash$ does not produce critical variables and therefore it also gives a type to e wrt. $\vdash^\bullet$.

Lemma 10.
If $\mathcal{A} \Vdash^\bullet e : \tau|\pi$ *then* $\Pi^e_{\mathcal{A}} = {}^\bullet\Pi^e_{\mathcal{A}}$.

Proof. From definition of $\Vdash^\bullet$ we know that $\mathcal{A} \Vdash^\bullet e : \tau|\pi$. We need to prove that $\Pi^e_\mathcal{A} \subseteq {}^\bullet\Pi^e_\mathcal{A}$ and ${}^\bullet\Pi^e_\mathcal{A} \subseteq \Pi^e_\mathcal{A}$.

$\Pi^e_\mathcal{A} \subseteq {}^\bullet\Pi^e_\mathcal{A}$) We prove that $\pi' \in \Pi^e_\mathcal{A} \implies \pi' \in {}^\bullet\Pi^e_\mathcal{A}$. If $\pi' \in \Pi^e_\mathcal{A}$ then $\mathcal{A}\pi' \vdash e : \tau'$, and by Theorem 5 there exists π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ and $\tau' = \tau\pi''$. By Theorem 4 $\mathcal{A}\pi \vdash^\bullet e : \tau$, and by Theorem 1-a $\mathcal{A}\pi\pi'' \vdash^\bullet e : \tau\pi''$, which is equal to $\mathcal{A}\pi' \vdash^\bullet e : \tau\pi''$; so $\pi' \in {}^\bullet\Pi^e_\mathcal{A}$.

${}^\bullet\Pi^e_\mathcal{A} \subseteq \Pi^e_\mathcal{A}$) From definition of ${}^\bullet\Pi^e_\mathcal{A}$

□

The following is a very simple lemma that states that from a set of type derivations $\vdash^\bullet$ for expressions $\overline{e_n}$ it is possible to create a type derivation $\vdash^\bullet$ for the tuple $(\overline{e_n})$, and vice versa (where the tuple constructor has the usual type $\forall\overline{\alpha_n}.\alpha_1 \to \ldots \alpha_n \to (\alpha_1, \ldots, \alpha_n)$).

Lemma 11.
$\mathcal{A} \vdash^\bullet e_1 : \tau_1, \ldots, \mathcal{A} \vdash^\bullet e_n : \tau_n \iff \mathcal{A} \vdash^\bullet (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n)$

Proof. Straightforward. □

Lemma 12.
$\mathcal{A} \vdash let_m\ X = e_2\ in\ e_1\ X : \tau \iff \mathcal{A} \vdash let_p\ X = e_2\ in\ e_1\ X : \tau$, *if* $X \notin var(e_1)$.

Proof. $\implies$) In this case we have a type derivation

$$\textbf{[LET}_m\textbf{]}\frac{\mathcal{A} \oplus \{X : \tau_x\} \vdash X : \tau_x \qquad \mathcal{A} \vdash e_2 : \tau_x \qquad \textbf{[APP]}\ \dfrac{\mathcal{A} \oplus \{X : \tau_x\} \vdash e_1 : \tau_x \to \tau \quad \mathcal{A} \oplus \{X : \tau_x\} \vdash X : \tau_x}{\mathcal{A} \oplus \{X : \tau_x\} \vdash e_1\ X : \tau}}{\mathcal{A} \vdash let_m\ X = e_2\ in\ e_1\ X : \tau}$$

It is clear that $Gen(\tau_x, \mathcal{A}) \succ \tau_x$ so $\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash X : \tau_x$, and by Theorem 1-d $\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_1 : \tau_x \to \tau$. Therefore the following type derivation is valid:

$$\textbf{[LET}_p\textbf{]}\frac{\mathcal{A} \oplus \{X : \tau_x\} \vdash X : \tau_x \qquad \mathcal{A} \vdash e_2 : \tau_x \qquad \textbf{[APP]}\ \dfrac{\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_1 : \tau_x \to \tau \quad \mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash X : \tau_x}{\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_1\ X : \tau}}{\mathcal{A} \vdash let_p\ X = e_2\ in\ e_1\ X : \tau}$$

$\Longleftarrow$) The type derivation is:

$$\textbf{[LET}_p\textbf{]}\frac{\mathcal{A} \oplus \{X : \tau_x\} \vdash X : \tau_x \qquad \mathcal{A} \vdash e_2 : \tau_x \qquad \textbf{[APP]}\ \dfrac{\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_1 : \tau' \to \tau \quad \mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash X : \tau'}{\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_1\ X : \tau}}{\mathcal{A} \vdash let_p\ X = e_2\ in\ e_1\ X : \tau}$$

By Lemma 3 we know that $\mathcal{A} \vdash e_2 : \tau'$. Clearly $\mathcal{A} \oplus \{X : \tau'\} \vdash X : \tau'$ is correct. As X does not appear in e_1 by Theorem 1-b the derivation $\mathcal{A} \oplus \{X : \tau'\} \vdash e_1 : \tau' \to \tau$ is valid. Therefore we can construct the following correct type derivation:

$$\textbf{[LET}_m\textbf{]}\frac{\mathcal{A} \oplus \{X : \tau'\} \vdash X : \tau' \qquad \mathcal{A} \vdash e_2 : \tau' \qquad \textbf{[APP]}\ \dfrac{\mathcal{A} \oplus \{X : \tau'\} \vdash e_1 : \tau' \to \tau \quad \mathcal{A} \oplus \{X : \tau'\} \vdash X : \tau'}{\mathcal{A} \oplus \{X : \tau'\} \vdash e_1\ X : \tau}}{\mathcal{A} \vdash let_p\ X = e_2\ in\ e_1\ X : \tau}$$

□

The first theorem states some properties of the typing relations $\vdash$ and $\vdash^\bullet$. Part *a)* states the closure under type substitutions. *b)* shows that type derivations for e depend only on the assumptions for the symbols in e. *c)* is a substitution lemma stating that in a type derivation we can replace a variable by an expression with the same type. Finally, *d)* establishes that from a valid type derivation we can change the assumption of a symbol for a more general type-scheme, and we still have a correct type derivation for the same type. Notice that this is not true wrt. the typing relation $\vdash^\bullet$ because a more general type can introduce opacity.

Theorem 1 (Properties of the typing relations).

a) If $\mathcal{A} \vdash^? e : \tau$ *then* $\mathcal{A}\pi \vdash^? e : \tau\pi$

b) Let s *be a symbol not appearing in* e*. Then* $\mathcal{A} \vdash^? e : \tau \iff \mathcal{A} \oplus \{s : \sigma_s\} \vdash^? e : \tau$.

c) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e : \tau$ *and* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e' : \tau_x$ *then* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e[X/e'] : \tau$.

d) If $\mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$ *and* $\sigma' \succ \sigma$*, then* $\mathcal{A} \oplus \{s : \sigma'\} \vdash e : \tau$.

Proof.
a.1) If $\mathcal{A} \vdash e : \tau$ *then* $\mathcal{A}\pi \vdash e : \tau\pi$
We prove it by induction over the size of the type derivation of $\mathcal{A} \vdash e : \tau$.

Base Case

- **[ID]** If we have a derivation of $\mathcal{A} \vdash s : \tau$ using **[ID]** is because τ is a generic instance of the type-scheme $\mathcal{A}(g) = \forall\overline{\alpha_n}.\tau'$. We can change this type-scheme by other equivalent $\forall\overline{\beta_n}.\tau''$ (according to Observation 1) where each variable β_i does not appear in $Dom(\pi)$ nor in $Rng(\pi)$. Then the generic instance τ will be of the form $\tau''[\overline{\beta_n/\tau_n}]$. We need to prove that $(\tau''[\overline{\beta_n/\tau_n}])\pi$ is a generic instance of $(\forall\overline{\beta_n}.\tau'')\pi$. Since π does not involve any variable β_i then $(\tau''[\overline{\beta_n/\tau_n}])\pi = \tau''\pi[\overline{\beta_n/\tau_n\pi}]$. Applying a substitution to a type-scheme is (by definition) applying it only to its free variables, but as no variable β_i appears in π then $(\forall\overline{\beta_n}.\tau'')\pi = \forall\overline{\beta_n}.(\tau''\pi)$. Then it is clear that $\tau''\pi[\overline{\beta_n/\tau_n\pi}]$ is a generic instance of $(\forall\overline{\beta_n}.\tau'')\pi$.

Induction Step
We have six different cases to consider accordingly to the inference rule used in the last step of the derivation.

- **[APP]** In this case we have a derivation

$$\textbf{[APP]}\ \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \qquad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

  By the Induction Hypothesis $\mathcal{A}\pi \vdash e_1 : (\tau_1 \to \tau)\pi$ and $\mathcal{A}\pi \vdash e_2 : \tau_1\pi$. $(\tau_1 \to \tau)\pi \equiv \tau_1\pi \to \tau\pi$ so we can construct a derivation

$$\textbf{[APP]}\ \frac{\mathcal{A}\pi \vdash e_1 : \tau_1\pi \to \tau\pi \qquad \mathcal{A}\pi \vdash e_2 : \tau_1\pi}{\mathcal{A} \vdash e_1\ e_2 : \tau\pi}$$

- **[Λ]** The derivation has the form

$$[\Lambda]\ \frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \qquad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau_t \to \tau}$$

  By the Induction Hypothesis $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\})\pi \vdash \lambda t : \tau_t\pi$ and $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\})\pi \vdash e : \tau\pi$. But $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\})\pi \equiv \mathcal{A}\pi \oplus (\{\overline{X_n : \tau_n}\})\pi \equiv \mathcal{A}\pi \oplus \{\overline{X_n : \tau_n\pi}\}$ so we can build the type derivation

$$[\Lambda]\ \frac{\mathcal{A}\pi \oplus \{\overline{X_n : \tau_n\pi}\} \vdash t : \tau_t\pi \qquad \mathcal{A}\pi \oplus \{\overline{X_n : \tau_n\pi}\} \vdash e : \tau\pi}{\mathcal{A}\pi \vdash \lambda t.e : \tau_t \to \tau\pi}$$

- **[LET$_m$]** The type derivation is

$$\textbf{[LET}_m\textbf{]}\ \frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \qquad \mathcal{A} \vdash e_1 : \tau_t \qquad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau}{\mathcal{A} \vdash let_m\ t = e_1\ in\ e_2 : \tau}$$

  By the Induction Hypothesis $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\})\pi \vdash t : \tau_t\pi$, $\mathcal{A}\pi \vdash e_1 : \tau_t\pi$ and $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\})\pi \vdash e_2 : \tau$. As in the previous case $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\})\pi \equiv \mathcal{A}\pi \oplus \{\overline{X_n : \tau_n\pi}\}$, so

$$\textbf{[LET}_m\textbf{]}\ \frac{\mathcal{A}\pi \oplus \{\overline{X_n : \tau_n\pi}\} \vdash t : \tau_t\pi \qquad \mathcal{A}\pi \vdash e_1 : \tau_t\pi \qquad \mathcal{A}\pi \oplus \{\overline{X_n : \tau_n\pi}\} \vdash e_2 : \tau\pi}{\mathcal{A}\pi \vdash let_m\ t = e_1\ in\ e_2 : \tau\pi}$$

- **[LET$^X_{pm}$]** The derivation will be

$$\textbf{[LET}^X_{pm}\textbf{]}\ \frac{\mathcal{A} \vdash e_1 : \tau_x \qquad \mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_2 : \tau}{\mathcal{A} \vdash let^X_{pm}\ X = e_1\ in\ e_2 : \tau}$$

  First, we create a substitution π' that maps the variables of τ_x which do not appear in $FTV(\mathcal{A})$ to fresh variables which are not in $FTV(\mathcal{A})$ and do not occur in $Dom(\pi)$ nor in $Rng(\pi)$. Then by the Induction Hypothesis $\mathcal{A}\pi' \vdash e_1 : \tau_x\pi'$. Since π' does not contain in its domain any variable in $FTV(\mathcal{A})$, then $\mathcal{A}\pi' = \mathcal{A}$ and $\mathcal{A} \vdash e_1 : \tau_x\pi'$. π' only substitutes variables which do not appear in $\mathcal{A}$ by variables which are not in $\mathcal{A}$ either, so $Gen(\tau_x, \mathcal{A}) = Gen(\tau_x\pi', \mathcal{A})$. Then $\mathcal{A} \oplus \{X : Gen(\tau_x\pi', \mathcal{A})\} \vdash e_2 : \tau$ is a valid derivation, and by the Induction Hypothesis

$(\mathcal{A} \oplus \{X : Gen(\tau_x\pi', \mathcal{A})\})\pi \vdash e_2 : \tau\pi$, which is the same that $\mathcal{A}\pi \oplus \{X : Gen(\tau_x\pi', \mathcal{A})\pi\} \vdash e_2 : \tau\pi$. By construction of π' we know that for every variable of $\tau_x\pi'$ which does not appear in $\mathcal{A}$ it will not be in $Dom(\pi)$ nor in $Rng(\pi)$. Then we can apply Lemma 8 and we have that $\mathcal{A}\pi \oplus \{X : Gen(\tau_x\pi'\pi, \mathcal{A}\pi)\} \vdash e_2 : \tau\pi$. By the Induction Hypothesis over $\mathcal{A} \vdash e_1 : \tau_x\pi'$ we obtain $\mathcal{A}\pi \vdash e_1 : \tau_x\pi'\pi$. With this information we can construct a derivation

$$\textbf{[LET}^X_{pm}\textbf{]}\ \frac{\mathcal{A}\pi \vdash e_1 : \tau_x\pi'\pi \qquad \mathcal{A}\pi \oplus \{X : Gen(\tau_x\pi'\pi, \mathcal{A}\pi)\} \vdash e_2 : \tau\pi}{\mathcal{A}\pi \vdash let^X_{pm}\ X = e_1\ in\ e_2 : \tau\pi}$$

- **[LET$^h_{pm}$]** Similar to the [LET$_m$] case.
- **[LET$_p$]** Similar to the [LET$^X_{pm}$] case, but instead of having to handle one single τ_x we need to handle a set of $\overline{\tau_n}$. The main idea is the same, creating a substitution π' to rename the variables of the $\overline{\tau_n}$ which do not appear in $\mathcal{A}$ and avoids their presence in the substitution π. Then we can apply Lemma 8 to all the generalizations and proceed as in the [LET$^X_{pm}$] case.

---

a.2) If $\mathcal{A} \vdash^\bullet e : \tau$ then $\mathcal{A}\pi \vdash^\bullet e : \tau\pi$

By definition of $\vdash^\bullet$ we know that $\mathcal{A} \vdash e : \tau$ and $critVar_{\mathcal{A}}(e) = \emptyset$. Then by Theorem 1-a $\mathcal{A}\pi \vdash e : \tau\pi$. To prove that $critVar_{\mathcal{A}\pi}(e) = \emptyset$ we use the decrease of opaque variables, stated in Lemma 5. From $\mathcal{A} \vdash e : \tau$ and $\mathcal{A}\pi \vdash e : \tau\pi$ we know that for every pattern t in e we have a derivation $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t$ and $\mathcal{A}\pi \oplus \{\overline{X_n : \tau'_n}\} \vdash t : \tau'$, being $\overline{X_n}$ the data variables in t. Then we can prove that $critVar_{\mathcal{A}\pi}(e) = \emptyset$ by induction over the structure of e.

Base Case

$s)$ $critVar_{\mathcal{A}\pi}(s) = \emptyset$ by definition.

Induction Step

- $e_1e_2)$ By the Induction Hypothesis we have that $critVar_{\mathcal{A}\pi}(e_1) = \emptyset$ and $critVar_{\mathcal{A}\pi}(e_2) = \emptyset$, so $critVar_{\mathcal{A}\pi}(e_1e_2) = critVar_{\mathcal{A}\pi}(e_1) \cup critVar_{\mathcal{A}\pi}(e_2) = \emptyset \cup \emptyset = \emptyset$.
- $\lambda t.e)$ By the Induction Hypothesis we have that $critVar_{\mathcal{A}\pi}(e) = \emptyset$. $critVar_{\mathcal{A}}(t) = \emptyset$, so $(opaqueVar_{\mathcal{A}}(t) \cap var(t)) = \emptyset$. By Lemma 5 we know that $opaqueVar_{\mathcal{A}\pi}(t) \subseteq opaqueVar_{\mathcal{A}}(t)$,

so $(opaqueVar_{\mathcal{A}\pi}(t) \cap var(t)) = \emptyset$. Therefore $critVar_{\mathcal{A}\pi}(\lambda t.e) = (opaqueVar_{\mathcal{A}\pi}(t) \cap var(t)) \cup critVar_{\mathcal{A}\pi}(e) = \emptyset \cup \emptyset = \emptyset$.

- $let_* \; t = e_1 \; in \; e_2$) Similar to the previous case.

---

b.1)Let be s a symbol which does not appear in e. Then $\mathcal{A} \vdash e : \tau \iff \mathcal{A} \oplus \{s : \sigma_s\} \vdash e : \tau$.

$\Longrightarrow$) We will proceed by induction over the size of the derivation tree.
Base Case

[ID] In this case the derivation will be:

$$\textbf{[ID]}\frac{}{\mathcal{A} \vdash s : \tau}$$

where $\mathcal{A}(g) \succ \tau$. If we add an assumption over a symbol different from s then $(\mathcal{A} \oplus \{s : \sigma_s\})(g) \succ \tau$, so

$$\textbf{[ID]}\frac{}{\mathcal{A} \oplus \{s : \sigma_s\} \vdash s : \tau}$$

Induction Step

[APP] The derivation will have the form:

$$\textbf{[APP]}\frac{\mathcal{A} \vdash e_1 : \tau' \to \tau \qquad \mathcal{A} \vdash e_2 : \tau'}{\mathcal{A} \vdash e_1 e_2 : \tau}$$

By the Induction Hypothesis then $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau' \to \tau$ and $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_2 : \tau'$, therefore:

$$\textbf{[APP]}\frac{\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau' \to \tau \qquad \mathcal{A} \oplus \{s : \sigma_s\} \vdash e_2 : \tau'}{\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 e_2 : \tau}$$

[Λ] We have a type derivation

$$\textbf{[}\Lambda\textbf{]}\frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau' \qquad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau' \to \tau}$$

By the Induction Hypothesis then $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{s : \sigma_s\} \vdash t : \tau'$ and $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{s : \sigma_s\} \vdash e : \tau$. s does not appear in $\lambda t.e$, so it will different from all the variables X_i and by Observation 3 $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{s : \sigma_s\}$ is the same as $(\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{\overline{X_n : \tau_n}\}$. Therefore we can build a type derivation:

$$[\Lambda]\frac{(\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau' \qquad (\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \oplus \{s : \sigma_s\} \vdash \lambda t.e : \tau' \to \tau}$$

[LET$_m$] The type derivation will be:

$$[\mathbf{LET}_m]\frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \qquad \mathcal{A} \vdash e_1 : \tau_t \qquad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau}{\mathcal{A} \vdash let_m\ t = e_1\ in\ e_2 : \tau}$$

By the Induction Hypothesis then $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{s : \sigma_s\} \vdash t : \tau_t$, $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau_t$ and $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{s : \sigma_s\} \vdash e : \tau$. As in the previous case $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{s : \sigma_s\} = (\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{\overline{X_n : \tau_n}\}$, so we can build a type derivation:

$$[\mathbf{LET}_m]\frac{\begin{array}{c}(\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \\ \mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau_t \\ (\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau\end{array}}{\mathcal{A} \oplus \{s : \sigma_s\} \vdash let_m\ t = e_1\ in\ e_2 : \tau}$$

[LET$^X_{pm}$] The type derivation will be:

$$[\mathbf{LET}^X_{pm}]\frac{\mathcal{A} \vdash e_1 : \tau_x \qquad \mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_2 : \tau}{\mathcal{A} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau}$$

Here the main problem is that $Gen(\tau_x, \mathcal{A})$ may not be same as $Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma_s\})$. This is caused because there are some type variables $\overline{\alpha_n}$ in $FTV(\tau_x)$ such that they appear free in $\mathcal{A}$ but not in $\mathcal{A} \oplus \{s : \sigma_s\}$ (they appear only in a previous assumption for s in $\mathcal{A}$) or because there are some type variables $\overline{\beta_n}$ in $FTV(\tau_x)$ such that they do not occur free in $\mathcal{A}$ but they do appear free in $\mathcal{A} \oplus \{s : \sigma_s\}$ (they are added by σ_s). The first group of variables will be generalized in $Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma_s\})$ but not in $Gen(\tau_x, \mathcal{A})$. To handle the second group we can create a type substitution π from $\overline{\beta_n}$ to fresh type variables. This way $Gen(\tau_x\pi, \mathcal{A} \oplus \{s : \sigma_s\})$ will be a type-scheme more general than $Gen(\tau_x, \mathcal{A})$, and by Theorem 1-d then $\mathcal{A} \oplus \{X : Gen(\tau_x\pi, \mathcal{A} \oplus \{s : \sigma_s\})\} \vdash e_2 : \tau$. By Theorem 1-a we obtain the derivation $\mathcal{A}\pi \vdash e_1 : \tau_x\pi$, and since $\overline{\beta_n}$ are not in $Dom(\pi)$ then $\mathcal{A} \vdash e_1 : \tau_x\pi$. By the Induction Hypothesis $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau_x\pi$ and $(\mathcal{A} \oplus \{X : Gen(\tau_x\pi, \mathcal{A} \oplus \{s : \sigma_s\})\}) \oplus \{s : \sigma_s\} \vdash e_2 : \tau$. As s is not in $let_m\ X = e_1\ in\ e_2$ then it is different from X, so $(\mathcal{A} \oplus \{X : Gen(\tau_x\pi, \mathcal{A} \oplus \{s : \sigma_s\})\}) \oplus \{s : \sigma_s\}$ is equal to $(\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{X : Gen(\tau_x\pi, \mathcal{A} \oplus \{s : \sigma_s\})\}$.

Therefore we can build the type derivation:

$$\textbf{[LET}^{X}_{pm}\textbf{]}\frac{\begin{array}{c}\mathcal{A}\oplus\{s:\sigma_s\}\vdash e_1:\tau_x\pi \\ (\mathcal{A}\oplus\{s:\sigma_s\})\oplus\{X:Gen(\tau_x\pi,\mathcal{A}\oplus\{s:\sigma_s\})\}\vdash e_2:\tau\end{array}}{\mathcal{A}\oplus\{s:\sigma_s\}\vdash let_{pm}\ X=e_1\ in\ e_2:\tau}$$

[LET$^{h}_{pm}$] Similar to the [LET$_m$] case.

[LET$_p$] Similar to the [LET$^{X}_{pm}$] case, creating a substitution π that solves the problem of the type variables which were generalized wrt. $\mathcal{A}$ but not wrt. $\mathcal{A}\oplus\{s:\sigma_s\}$.

$\Longleftarrow$) We will proceed again by induction over the size of the derivation tree.

Base Case

When the type derivation only applies the [ID] rule the proof is straightforward.

Induction Step

[APP] The derivation will have the form:

$$\textbf{[APP]}\frac{\mathcal{A}\oplus\{s:\sigma_s\}\vdash e_1:\tau'\to\tau \qquad \mathcal{A}\oplus\{s:\sigma_s\}\vdash e_2:\tau'}{\mathcal{A}\oplus\{s:\sigma_s\}\vdash e_1e_2:\tau}$$

By the Induction Hypothesis then $\mathcal{A}\vdash e_1:\tau'\to\tau$ and $\mathcal{A}\vdash e_2:\tau'$, therefore:

$$\textbf{[APP]}\frac{\mathcal{A}\vdash e_1:\tau'\to\tau \qquad \mathcal{A}\vdash e_2:\tau'}{\mathcal{A}\vdash e_1e_2:\tau}$$

[Λ] We have the type derivation:

$$\textbf{[}\Lambda\textbf{]}\frac{(\mathcal{A}\oplus\{s:\sigma_s\})\oplus\{\overline{X_n:\tau_n}\}\vdash t:\tau' \qquad (\mathcal{A}\oplus\{s:\sigma_s\}))\oplus\{\overline{X_n:\tau_n}\}\vdash e:\tau}{\mathcal{A}\oplus\{s:\sigma_s\}\vdash\lambda t.e:\tau'\to\tau}$$

Since s is not in $\lambda t.e$, s will be different from all the variables $\overline{X_n}$ and $(\mathcal{A}\oplus\{s:\sigma_s\})\oplus\{\overline{X_n:\tau_n}\}$ will be the same as $(\mathcal{A}\oplus\{\overline{X_n:\tau_n}\})\oplus\{s:\sigma_s\}$. Having $(\mathcal{A}\oplus\{\overline{X_n:\tau_n}\})\oplus\{s:\sigma_s\}\vdash t:\tau'$ and $(\mathcal{A}\oplus\{\overline{X_n:\tau_n}\})\oplus\{s:\sigma_s\}\vdash e:\tau$ we can apply the Induction Hypothesis and obtain $\mathcal{A}\oplus\{\overline{X_n:\tau_n}\}\vdash t:\tau'$ and $\mathcal{A}\oplus\{\overline{X_n:\tau_n}\}\vdash e:\tau$. With these two derivation we can build:

$$\textbf{[}\Lambda\textbf{]}\frac{\mathcal{A}\oplus\{\overline{X_n:\tau_n}\}\vdash t:\tau' \qquad \mathcal{A}\oplus\{\overline{X_n:\tau_n}\}\vdash e:\tau}{\mathcal{A}\vdash\lambda t.e:\tau'\to\tau}$$

[LET$_m$] Similar to the [Λ] case.

[LET$^X_{pm}$] This case has to deal with the same problems as in [LET$^X_{pm}$] of the $\Longrightarrow$) case. We have a type derivation:

$$\textbf{[LET}^X_{pm}\textbf{]}\frac{\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau_x \qquad (\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma_s\})\} \vdash e_2 : \tau}{\mathcal{A} \oplus \{s : \sigma_s\} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau}$$

Again, the problem is that $Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma_s\})$ may not be the same as $Gen(\tau_x, \mathcal{A})$. As before, there may be variables $\overline{\alpha_n}$ in $FTV(\tau_x)$ which appear free in $\mathcal{A} \oplus \{s : \sigma_s\}$ but not in $\mathcal{A}$, and variables $\overline{\beta_n}$ in $FTV(\tau_x)$ which do not occur free in $\mathcal{A} \oplus \{s : \sigma_s\}$ but they do appear free in $\mathcal{A}$. The first group is not problematic, because they are variables which will be generalized in $Gen(\tau_x, \mathcal{A})$ but not in $Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma_s\})$. To solve the problem with the second group we create a type substitution π from $\overline{\beta}$ to fresh variables. This way $Gen(\tau_x\pi, \mathcal{A})$ will be a more general type-scheme than $Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma_s\})$. Applying Theorem 1-d then $(\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{X : Gen(\tau_x\pi, \mathcal{A})\} \vdash e_2 : \tau$. As s is different from X, then $(\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{X : Gen(\tau_x\pi, \mathcal{A})\}$ is the same as $(\mathcal{A} \oplus \{X : Gen(\tau_x\pi, \mathcal{A})\}) \oplus \{s : \sigma_s\}$, so the derivation $(\mathcal{A} \oplus \{X : Gen(\tau_x\pi, \mathcal{A})\}) \oplus \{s : \sigma_s\} \vdash e_2 : \tau$ is correct. Applying the Induction Hypothesis to this derivation we obtain $\mathcal{A} \oplus \{X : Gen(\tau_x\pi, \mathcal{A})\} \vdash e_2 : \tau$. By Theorem 1-a $(\mathcal{A} \oplus \{s : \sigma_s\})\pi \vdash e_1 : \tau_x\pi$, which is equal to $\mathcal{A} \oplus \{s : \sigma_s\pi\} \vdash e_1 : \tau_x\pi$ because $\overline{\beta_n}$ do not occur free in $\mathcal{A}$. Applying the Induction Hypothesis to this derivation, we obtain $\mathcal{A} \vdash e_1 : \tau_x\pi$. Therefore we can build the type derivation:

$$\textbf{[LET}^X_{pm}\textbf{]}\frac{\mathcal{A} \vdash e_1 : \tau_x\pi \qquad \mathcal{A} \oplus \{X : Gen(\tau_x\pi, \mathcal{A})\} \vdash e_2 : \tau}{\mathcal{A} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau}$$

[LET$^h_{pm}$] Similar to the [Λ] case.

[LET$_p$] Similar to the [LET$^X_{pm}$] case.

---

b.2) Let be s a symbol which does not appear in e, and σ_s any type. Then $\mathcal{A} \vdash^\bullet e : \tau \Longleftrightarrow \mathcal{A} \oplus \{s : \sigma_s\} \vdash^\bullet e : \tau$.

$\Longrightarrow$) By definition of $\mathcal{A} \vdash^\bullet e : \tau$, $\mathcal{A} \vdash e : \tau$ and $critVar_{\mathcal{A}}(e) = \emptyset$. Since s does not occur in e by Theorem 1-b $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e : \tau$. It will also be true that $critVar_{\mathcal{A} \oplus \{s:\sigma_s\}}(e) = \emptyset$ because the opaque variables in the patterns will not change by adding the new assumption, and neither the variables appearing in the rest of the expression. Therefore $\mathcal{A} \oplus \{s : \sigma_s\} \vdash^\bullet e : \tau$.

$\Longleftarrow$) By definition of $\mathcal{A} \oplus \{s : \sigma_s\} \vdash^\bullet e : \tau$, $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e : \tau$ and $critVar_{\mathcal{A}\oplus\{s:\sigma_s\}}(e) = \emptyset$. s does not appear in e, so by Theorem 1-b $\mathcal{A} \vdash e : \tau$. As in the previous case the critical variables of e will not change by deleting an assumption which is not used, so $\mathcal{A} \vdash^\bullet e : \tau$.

---

c.1) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash e : \tau$ *and* $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$ *then* $\mathcal{A} \oplus \{X : \tau_x\} \vdash e[X/e'] : \tau$.

We will proceed by induction over the size of the expression e.

Base Case

[ID] If $s \neq X$ then $s[X/e'] \equiv s$. On the contrary, if $s = X$ then the derivation will be:

$$\textbf{[ID]}\frac{}{\mathcal{A} \oplus \{X : \tau_x\} \vdash X : \tau_x}$$

$X[X/e'] \equiv e'$, and the type derivation $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$ comes from the hypothesis.

Induction Step

[APP] Just the application of the Induction Hypothesis.

[Λ] We can assume that $\lambda t.e$ is such that the variables $\overline{X_n}$ in its pattern do not appear in $\mathcal{A} \oplus \{X : \tau_x\}$ nor in $FV(e')$. The derivation will have the form:

$$[\Lambda]\frac{(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau' \qquad (\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \oplus \{X : \tau_x\} \vdash \lambda t.e : \tau' \to \tau}$$

As X is different from $\overline{X_n}$ then $(\lambda t.e)[X/e'] \equiv \lambda t.(e[X/e'])$, so the first derivation remains the same. We have from the hypothesis that $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$. Since none of the $\overline{X_n}$ appear in e' then by Theorem 1-b we can add assumptions over that variables and obtain a derivation $(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{\overline{X_n : \tau_n}\} \vdash e' : \tau_x$. Because $X \neq X_i$ for all i then by Observation 3 $(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{\overline{X_n : \tau_n}\}$ is the same as $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{X : \tau_x\}$. We have $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{X : \tau_x\} \vdash e : \tau$ and $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{X : \tau_x\} \vdash e' : \tau_x$, so applying the Induction Hypothesis we obtain $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{X : \tau_x\} \vdash e[X/e'] : \tau$. Therefore we can build a new derivation:

$$[\Lambda]\frac{(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau' \qquad (\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{\overline{X_n : \tau_n}\} \vdash e[X/e'] : \tau}{\mathcal{A} \oplus \{X : \tau_x\} \vdash \lambda t.(e[X/e']) : \tau' \to \tau}$$

[LET$_m$] The proof is similar to the [Λ] case, provided that the variables of the pattern t do not occur in $FV(e')$ nor in $\mathcal{A} \oplus \{X : \tau_x\}$.

[LET$^X_{pm}$] In this case Y is a fresh variable. The type derivation will be:

$$[\textbf{LET}^X_{pm}]\frac{\mathcal{A} \oplus \{X : \tau_x\} \vdash e_1 : \tau_x \qquad (\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\} \vdash e_2 : \tau}{\mathcal{A} \oplus \{X : \tau_x\} \vdash let_{pm}\ Y = e_1\ in\ e_2 : \tau}$$

By the Induction Hypothesis $\mathcal{A} \oplus \{X : \tau_x\} \vdash e_1[X/e'] : \tau_x$. $X \neq Y$ and $Y \notin FV(e')$, so by Theorem 1-b we can add an assumption over the variable Y and get a derivation $(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\} \vdash e' : \tau_x$. By Observation 3 $(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\}$ is equal to $(\mathcal{A} \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\}) \oplus \{X : \tau_x\}$, so by the Induction Hypothesis $(\mathcal{A} \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\}) \oplus \{X : \tau_x\} \vdash e_2[X/e'] : \tau$. Again by Observation 3 $(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\} \vdash e_2[X/e'] : \tau$. Therefore we can construct a derivation:

$$[\textbf{LET}^X_{pm}]\frac{\mathcal{A} \oplus \{X : \tau_x\} \vdash e_1[X/e'] : \tau_x \qquad (\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\} \vdash e_2[X/e'] : \tau}{\mathcal{A} \oplus \{X : \tau_x\} \vdash let_{pm}\ Y = e_1[X/e']\ in\ e_2[X/e'] : \tau}$$

[LET$^h_{pm}$] Equal to the [LET$_m$] case.

[LET$_p$] The proof follows the same ideas as [LET$_m$] and [LET$^X_{pm}$].

---

c.2) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash^\bullet e : \tau$ and $\mathcal{A} \oplus \{X : \tau_x\} \vdash^\bullet e' : \tau_x$ then $\mathcal{A} \oplus \{X : \tau_x\} \vdash^\bullet e[X/e'] : \tau$.

From the definition of $\vdash^\bullet$ we know that $\mathcal{A} \oplus \{X : \tau_x\} \vdash e : \tau$, $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$, $critVar_{\mathcal{A} \oplus \{X:\tau_x\}}(e) = \emptyset$ and $critVar_{\mathcal{A} \oplus \{X:\tau_x\}}(e') = \emptyset$. Then by Theorem 1-c $\mathcal{A} \oplus \{X : \tau_x\} \vdash e[X/e'] : \tau$. By Lemma 7 we also know that $critVar_{\mathcal{A} \oplus \{X:\tau_x\}}(e[X/e']) = \emptyset$, so by definition $\mathcal{A} \oplus \{X : \tau_x\} \vdash^\bullet e[X/e'] : \tau$.

---

d.1) If $\mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$ *and* $\sigma' \succ \sigma$*, then* $\mathcal{A} \oplus \{s : \sigma'\} \vdash e : \tau$.

Base Case

[ID] If $e \neq s$ then is trivial. If $e = s$ then the derivation will be:

$$\textbf{[ID]}\frac{}{\mathcal{A} \oplus \{s : \sigma\} \vdash s : \tau}$$

where $\sigma \succ \tau$. By Definition of generic instance, since $\sigma' \succ \sigma$ then $\sigma' \succ \tau$. So we can build the derivation:

$$\textbf{[ID]}\frac{}{\mathcal{A} \oplus \{s : \sigma'\} \vdash s : \tau}$$

Induction Step

[APP] We have a type derivation:

$$\textbf{[APP]}\frac{\mathcal{A} \oplus \{s : \sigma\} \vdash e_1 : \tau' \to \tau \qquad \mathcal{A} \oplus \{s : \sigma\} \vdash e_2 : \tau'}{\mathcal{A} \oplus \{s : \sigma\} \vdash e_1 e_2 : \tau}$$

By the Induction Hypothesis we have that $\mathcal{A} \oplus \{s : \sigma'\} \vdash e_1 : \tau' \to \tau$ and $\mathcal{A} \oplus \{s : \sigma'\} \vdash e_2 : \tau'$. Then we can construct a type derivation with the more general assumptions:

$$\textbf{[APP]}\frac{\mathcal{A} \oplus \{s : \sigma'\} \vdash e_1 : \tau' \to \tau \qquad \mathcal{A} \oplus \{s : \sigma'\} \vdash e_2 : \tau'}{\mathcal{A} \oplus \{s : \sigma'\} \vdash e_1 e_2 : \tau}$$

[Λ] We can assume that s is different from all the variables $\overline{X_n}$. The type derivation will be:

$$[\Lambda]\frac{(\mathcal{A} \oplus \{s : \sigma\}) \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau' \qquad (\mathcal{A} \oplus \{s : \sigma\}) \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \oplus \{s : \sigma\} \vdash \lambda t.e : \tau' \to \tau}$$

Since s is different from the variables $\overline{X_n}$, then $(\mathcal{A} \oplus \{s : \sigma\}) \oplus \{\overline{X_n : \tau_n}\}$ is the same as $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{s : \sigma\}$. Therefore $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{s : \sigma\} \vdash t : \tau'$ and $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{s : \sigma\} \vdash e : \tau$. By the Induction Hypothesis we have that $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{s : \sigma'\} \vdash t : \tau'$ and $(\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}) \oplus \{s : \sigma'\} \vdash e : \tau$; and changing again the order in the assumptions we can build a derivation:

$$[\Lambda]\frac{(\mathcal{A} \oplus \{s : \sigma'\}) \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau' \qquad (\mathcal{A} \oplus \{s : \sigma'\}) \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \oplus \{s : \sigma'\} \vdash \lambda t.e : \tau' \to \tau}$$

[LET$_m$] The proof is similar to the [Λ] case.

[LET$^X_{pm}$] We assume that $s \neq X$. The type derivation is:

$$\textbf{[LET}^X_{pm}\textbf{]}\frac{\mathcal{A} \oplus \{s : \sigma\} \vdash e_1 : \tau_x \quad (\mathcal{A} \oplus \{s : \sigma\}) \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma\})\} \vdash e_2 : \tau}{\mathcal{A} \oplus \{s : \sigma\} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau}$$

By the Induction Hypothesis we have $\mathcal{A} \oplus \{s : \sigma'\} \vdash e_1 : \tau_x$. As $\sigma' \succ \sigma$ then by Observation 2 $FTV(\sigma') \subseteq FTV(\sigma)$. Therefore $FTV(\mathcal{A} \oplus \{s : \sigma'\}) = FTV(\mathcal{A}_s) \cup FTV(\sigma') \subseteq FTV(\mathcal{A}_s) \cup FTV(\sigma) = FTV(\mathcal{A} \oplus \{s : \sigma\})$, being $\mathcal{A}_s$ the result of deleting from $\mathcal{A}$ all the assumptions for the symbol s. With this information it is clear that $Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\}) \succ Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma\})$ because more variables could be generalized in $Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\})$. Then by the Induction Hypothesis $(\mathcal{A} \oplus \{s : \sigma\}) \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\})\} \vdash e_2 : \tau$. As $s \neq X$ then we can change the order of the assumptions and obtain a derivation $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\})\}) \oplus \{s : \sigma\} \vdash e_2 : \tau$. Again by the Induction Hypothesis $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\})\}) \oplus \{s : \sigma'\} \vdash e_2 : \tau$. With these derivations we can build the one we were trying to construct:

$$\textbf{[LET}^X_{pm}\textbf{]}\frac{\mathcal{A} \oplus \{s : \sigma'\} \vdash e_1 : \tau_x \quad (\mathcal{A} \oplus \{s : \sigma'\}) \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\})\} \vdash e_2 : \tau}{\mathcal{A} \oplus \{s : \sigma'\} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau}$$

[LET$^h_{pm}$] Similar to the [Λ] case.

[LET$_p$] The proof is similar to the [LET$^X_{pm}$] case.

□

The following theorem states that the transformation $TRL(e)$ for eliminating compound patterns in let expressions of Figure 3.4 preserves the type of the expression. It also states that the projection functions created in the transformation are well-typed wrt. the original set of assumptions extended with the assumptions for those functions.

Theorem 2 (Type preservation of the let transformation).
Assume $\mathcal{A} \vdash^\bullet e : \tau$ and let $\mathcal{P} \equiv \{\overline{f_{X_n}\ t_n \rightarrow X_n}\}$ be the rules of the projection functions needed in the transformation of e according to Figure 3.4. Let also $\mathcal{A}'$ be the set of assumptions over that functions, defined as $\mathcal{A}' \equiv \{\overline{f_{X_n} : Gen(\tau_{X_n}, \mathcal{A})}\}$, where $\mathcal{A} \Vdash^\bullet \lambda t_i.X_i : \tau_{X_i} | \pi_{X_i}$. Then $\mathcal{A} \oplus \mathcal{A}' \vdash^\bullet TRL(e) : \tau$ and $wt_{\mathcal{A} \oplus \mathcal{A}'}(\mathcal{P})$.

Proof. By structural induction over the expression e.

Base Case

- s) Straightforward.

Induction Step

- $e_1\ e_2$) We have the type derivation:

$$\textbf{[APP]}\frac{\mathcal{A} \vdash e_1 : \tau_1 \rightarrow \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

Let be $\mathcal{A}^1$ and $\mathcal{A}^2$ the assumptions over the projection functions needed in e_1 and e_2 respectively. The by the Induction Hypothesis $\mathcal{A} \oplus \mathcal{A}^1 \vdash TRL(e_1)$ and $\mathcal{A} \oplus \mathcal{A}^2 \vdash TRL(e_2)$. Clearly the set of assumptions $\mathcal{A}'$ over the projection functions needed in the whole expression is $\mathcal{A}^1 \oplus \mathcal{A}^2$. Then by Theorem 1-b both derivations $\mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_1)$ and $\mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_2)$ are valid, and we can construct the type derivation:

$$\textbf{[APP]}\frac{\mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_1) : \tau_1 \rightarrow \tau \quad \mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_2) : \tau_1}{\mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_1)\ TRL(e_2) : \tau}$$

- $let_K\ X = e_1\ in\ e_2$) There are two cases, depending on the K:
  $let_m\ X = e_1\ in\ e_2$:
  The type derivation will be

$$\textbf{[LET}_m\textbf{]}\frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{X : \tau_t\} \vdash e_2 : \tau}{\mathcal{A} \vdash let_m\ X = e_1\ in\ e_2 : \tau}$$

By the Induction Hypothesis $\mathcal{A} \vdash TRL(e_1) : \tau_t$ and $\mathcal{A} \oplus \{X : \tau_t\} \vdash TRL(e_2) : \tau$. Then we can build the type derivation

$$\textbf{[LET}_m\textbf{]}\frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \quad \mathcal{A} \vdash TRL(e_1) : \tau_t \quad \mathcal{A} \oplus \{X : \tau_t\} \vdash TRL(e_2) : \tau}{\mathcal{A} \vdash let_m\ X = TRL(e_1)\ in\ TRL(e_2) : \tau}$$

  $let_p\ X = e_1\ in\ e_2$:
  The type derivation for the original expression is

$$\textbf{[LET}_m\textbf{]}\frac{\begin{array}{c}\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \\ \mathcal{A} \vdash e_1 : \tau_t \\ \mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\} \vdash e_2 : \tau\end{array}}{\mathcal{A} \vdash let_p\ X = e_1\ in\ e_2 : \tau}$$

By the Induction Hypothesis $\mathcal{A} \vdash TRL(e_1) : \tau_t$ and $\mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\} \vdash TRL(e_2) : \tau$. Then we can build the type derivation

$$\textbf{[LET}_m\textbf{]}\frac{\begin{array}{c}\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \\ \mathcal{A} \vdash TRL(e_1) : \tau_t \\ \mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\} \vdash TRL(e_2) : \tau\end{array}}{\mathcal{A} \vdash let_p\ X = TRL(e_1)\ in\ TRL(e_2) : \tau}$$

- $let_{pm}\ X = e_1\ in\ e_2$) The type derivation for the original expression is

$$\textbf{[LET}_{pm}\textbf{]}\frac{\begin{array}{c}\mathcal{A} \vdash e_1 : \tau_t \\ \mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\} \vdash e_2 : \tau\end{array}}{\mathcal{A} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau}$$

By the Induction Hypothesis $\mathcal{A} \vdash TRL(e_1) : \tau_t$ and $\mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\} \vdash TRL(e_2) : \tau$. The type derivation $\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t$ is trivial, so we can build the type derivation

$$\textbf{[LET}_m\textbf{]}\frac{\begin{array}{c}\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \\ \mathcal{A} \vdash TRL(e_1) : \tau_t \\ \mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\} \vdash TRL(e_2) : \tau\end{array}}{\mathcal{A} \vdash let_p\ X = TRL(e_1)\ in\ TRL(e_2) : \tau}$$

- $let_m\ t = e_1\ in\ e_2$) In this case the original type derivation is:

$$\textbf{[LET}_m\textbf{]}\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \\ \mathcal{A} \vdash e_1 : \tau_t \\ \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau\end{array}}{\mathcal{A} \vdash let_m\ t = e_1\ in\ e_2 : \tau}$$

It is easy to see that if $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t$ then $\mathcal{A} \vdash \lambda t.X_i : \tau_t \to \tau_i$. The assumptions over the projections functions in $\mathcal{A}'$ will be $\{\overline{f_{X_n} : Gen(\tau'_t \to \tau'_n, \mathcal{A})}\}$, where $\mathcal{A} \Vdash \lambda t.X_i : \tau'_t \to \tau'_i | \pi_{X_i}$. Since $\mathcal{A} \vdash \lambda t.X_i : \tau_t \to \tau_i$ we can assume that $\mathcal{A}\pi_{X_i} = \mathcal{A}$ (Observation 6), and by Theorem 5 we know that exists a type substitution π such that $\mathcal{A}\pi_{X_i}\pi = \mathcal{A}\pi = \mathcal{A}$ and $(\tau'_t \to \tau'_i)\pi = \tau_t \to \tau_i$. Therefore we can be sure that $Gen(\tau'_t \to \tau'_i, \mathcal{A}) \succ \tau_t \to \tau_i$, because π substitutes only the type variables in $\tau'_t \to \tau'_i$ which are generalized in $Gen(\tau'_t \to \tau'_i, \mathcal{A})$. If $\mathcal{A}'$ contains all the assumptions over the projection functions needed in the whole expression, it will contains assumptions over projection functions needed in e_1 ($\mathcal{A}^1$), e_2 ($\mathcal{A}^2$) and the pattern t ($\mathcal{A}^t \equiv \{\overline{f_{X_n} : Gen(\tau'_t \to \tau'_n, \mathcal{A})}\}$); so $\mathcal{A}' = \mathcal{A}^1 \oplus \mathcal{A}^2 \oplus \mathcal{A}^t$. Then we can build the type derivation:

$$\textbf{[LET}_m\textbf{]}\frac{\mathcal{A} \oplus \mathcal{A}' \oplus \{Y : \tau_t\} \vdash Y : \tau_t \quad \mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_1) : \tau_t \quad \mathcal{A}_Y \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ TRL(e_2) : \tau}{\mathcal{A} \oplus \mathcal{A}' \vdash let_m\ Y = TRL(e_1)\ in\ \overline{let_m\ X_n = f_{X_n}\ Y}\ in\ TRL(e_2) : \tau}$$

where the derivation $\mathcal{A}_Y \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ TRL(e_2) : \tau$ is

$$\textbf{[LET}_m\textbf{]}\frac{\mathcal{A}_Y \oplus \{X_1 : \tau_1\} \vdash X_1 : \tau_1 \quad \textbf{[APP]}\ \frac{\mathcal{A}_Y \vdash f_{X_1} : \tau_t \to \tau_1 \quad \mathcal{A}_Y \vdash Y : \tau_t}{\mathcal{A}_Y \vdash f_{X_1}\ Y : \tau_1} \quad \textbf{[LET}_m\textbf{]}\ \frac{\mathcal{A}_Y \oplus \{\overline{X_n : \tau_n}\} \vdash TRL(e_2) : \tau}{\ldots}}{\mathcal{A}_Y \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ TRL(e_2) : \tau}$$

(being $\mathcal{A}_Y \equiv \mathcal{A} \oplus \mathcal{A}' \oplus \{Y : \tau_t\}$).

$\mathcal{A} \oplus \mathcal{A}' \oplus \{Y : \tau_t\} \vdash Y : \tau_t$ and $\mathcal{A}_Y \oplus \{X_1 : \tau_1\} \vdash X_1 : \tau_1$ are just the application of **[ID]** rule. By the Induction Hypothesis $\mathcal{A} \oplus \mathcal{A}^1 \vdash TRL(e_1) : \tau_t$, and by Theorem 1-b we can add the assumptions $\mathcal{A}^2 \oplus \mathcal{A}^t$, obtaining $\mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_1) : \tau_t$. $\mathcal{A}_Y \vdash f_{X_1}\ Y : \tau_1$ is straightforward because $Gen(\tau'_t \to \tau'_i, \mathcal{A}\pi_{X_i}) \succ \tau_t \to \tau_i$ for all the projection functions. It is easy to see that this way the chain of let expressions will "collect" the same assumptions for the variables $\overline{X_n}$ that are introduced by the pattern in the original expression: $\{\overline{X_n : \tau_n}\}$. Then by the Induction Hypothesis $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \oplus \mathcal{A}^2 \vdash TRL(e_2) : \tau$, and by Theorem 1-b we can add the rest of the assumptions and obtain $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \oplus \mathcal{A}^2 \oplus \mathcal{A}^1 \oplus \mathcal{A}^t \oplus \{Y : \tau_t\} \vdash TRL(e_2) : \tau$. Reorganizing the set of assumptions (since the symbols are all different), we obtain $\mathcal{A}_Y \oplus \{\overline{X_n : \tau_n}\} \vdash TRL(e_2) : \tau$.

- $let_{pm}\ t = e_1\ in\ e_2$) This case is equal to the previous one because the derivation of the original expression in both cases is the same (as t is a pattern we use **[LET**$^h_{pm}$**]**, and this rule acts equal to **[LET**$_m$**]**) and the transformed expressions are the same.

- $let_p\ t = e_1\ in\ e_2$) The type derivation will be:

$$\textbf{[LET}_p\textbf{]}\frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : Gen(\tau_n, \mathcal{A})}\} \vdash e_2 : \tau}{\mathcal{A} \vdash let_p\ t = e_1\ in\ e_2 : \tau}$$

As in the previous case, $\mathcal{A}'$ will be $\{\overline{f_{X_n} : Gen(\tau'_t \to \tau'_n, \mathcal{A}\pi_{X_n})}\}$, where $\mathcal{A} \Vdash \lambda t.X_i : \tau'_t \to \tau'_i|\pi_{X_i}$. In addition, $\mathcal{A}\pi_{X_i} = \mathcal{A}$ (Observation 6), $Gen(\tau'_t \to \tau'_i, \mathcal{A}) \succ \tau_t \to \tau_i$ and $\mathcal{A}' \equiv \mathcal{A}^1 \oplus \mathcal{A}^2 \oplus \mathcal{A}^t$. Then we can build a type derivation:

$$\textbf{[LET}_p\textbf{]}\frac{\mathcal{A} \oplus \mathcal{A}' \oplus \{Y : \tau_t\} \vdash Y : \tau_t \quad \mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_1) : \tau_t \quad \mathcal{A}'_1 \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ TRL(e_2) : \tau}{\mathcal{A} \oplus \mathcal{A}' \vdash let_p\ Y = TRL(e_1)\ in\ \overline{let_p\ X_n = f_{X_n}\ Y}\ in\ TRL(e_2) : \tau}$$

where the derivation $\mathcal{A}_Y \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ TRL(e_2) : \tau$ is

$$\textbf{[LET}_p\textbf{]}\ \frac{\mathcal{A}'_1 \oplus \{X_1 : \tau_1\} \vdash X_1 : \tau_1 \qquad \textbf{[APP]}\ \dfrac{\mathcal{A}'_1 \vdash f_{X_1} : \tau_t \to \tau_1 \quad \mathcal{A}'_1 \vdash Y : \tau_t}{\mathcal{A}'_1 \vdash f_{X_1}\ Y : \tau_1} \qquad \textbf{[LET}_p\textbf{]}\ \dfrac{\textbf{[LET}_p\textbf{]}\ \dfrac{\mathcal{A}'_{n+1} \vdash TRL(e_2) : \tau}{\ldots}}{\mathcal{A}'_2 \vdash let_p\ X_2 = f_{X_2}\ Y\ in \ldots in\ TRL(e_2)}}{\mathcal{A}'_1 \vdash let_p\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ TRL(e_2) : \tau}$$

being $\mathcal{A}'_1 \equiv \mathcal{A} \oplus \mathcal{A}' \oplus \{Y : Gen(\tau_t, \mathcal{A} \oplus \mathcal{A}')\}$ and $\mathcal{A}'_i \equiv \mathcal{A}'_{i-1} \oplus \{X_{i-1} : Gen(\tau_{i-1}, \mathcal{A}'_{i-1})\}$.

As in the previous case, all the derivations $\mathcal{A}'_i \vdash f_{X_i}\ Y : \tau_i$ are valid, because $\mathcal{A}'_i \vdash Y : \tau_t$. Notice that $Gen(\tau_t, \mathcal{A}) = Gen(\tau_t, \mathcal{A} \oplus \mathcal{A}')$, as Observation 4 states, since $FTV(\mathcal{A}) = FTV(\mathcal{A} \oplus \mathcal{A}')$. For the same reason, $Gen(\tau_i, \mathcal{A}) = Gen(\tau_i, \mathcal{A}'_i)$, so the chain of let expressions will collect the same set of assumptions over the variables $\overline{X_n}$: $\{\overline{X_n : Gen(\tau_n, \mathcal{A})}\}$. By the Induction Hypothesis, we know that $\mathcal{A} \oplus \{\overline{X_n : Gen(\tau_n, \mathcal{A})}\} \oplus \mathcal{A}^2 \vdash TRL(e_2) : \tau$; and by Theorem 1-b we can add the assumptions $\mathcal{A}^1 \oplus \mathcal{A}^t \oplus \{Y : Gen(\tau_t, \mathcal{A} \oplus \mathcal{A}')$ and obtain $\mathcal{A} \oplus \{\overline{X_n : Gen(\tau_n, \mathcal{A})}\} \oplus \mathcal{A}^2 \oplus \mathcal{A}^1 \oplus \mathcal{A}^t \oplus \{Y : Gen(\tau_t, \mathcal{A} \oplus \mathcal{A}')\} \vdash TRL(e_2) : \tau$. Then reorganizing the assumptions we obtain $\mathcal{A} \oplus \mathcal{A}' \oplus \{Y : Gen(\tau_t, \mathcal{A} \oplus \mathcal{A}')\} \oplus \{\overline{X_n : Gen(\tau_n, \mathcal{A})}\} \vdash TRL(e_2) : \tau$. Since $Gen(\tau_i, \mathcal{A}) = Gen(\tau_i, \mathcal{A}'_i)$ then the previous derivation is equal to $\mathcal{A}'_{n+1} \vdash TRL(e_2) : \tau$.

In all the cases it is true that $wt_{\mathcal{A} \oplus \mathcal{A}'}(\mathcal{P})$. Let X_i a data variable which is projected in the transformed expression, and t_i the compound pattern of a let expression where it appears. By Observation 7 we know that in the derivation $\mathcal{A} \vdash^{\bullet} e : \tau$ will appear a derivation $\mathcal{A} \oplus \mathcal{A}'' \oplus \{X_i : \tau'_{X_i}\} \vdash t_i : \tau_i$ for a set of assumptions $\mathcal{A}''$ over some variables and X_i will not be opaque in t_i wrt. $\mathcal{A} \oplus \mathcal{A}'' \oplus \{X_i : \tau'_{X_i}\}$. Then it is clear that $\mathcal{A} \vdash \lambda t_i.X_i : \tau_i \to \tau'_{X_i}$, and by Theorem 5 the type inference $\mathcal{A} \Vdash \lambda t_i.X_i : \tau_{X_i} | \pi_{X_i}$ will be correct. By Theorem 4 $\mathcal{A}\pi_{X_i} \vdash \lambda t_i.X_i : \tau_{X_i}$, and since by Observation 3 $\mathcal{A}\pi_{X_i} = \mathcal{A}$, then $\mathcal{A} \vdash \lambda t_i.X_i : \tau_{X_i}$ is a valid derivation. Clearly X_i is not opaque in t_i wrt. $\mathcal{A}$, because only the assumptions for non variable symbols are used. Then $critVar_{\mathcal{A}}(\lambda t_i.X_i) = \emptyset$, so $\mathcal{A} \Vdash^{\bullet} \lambda t_i.X_i : \tau_{X_i} | \pi_{X_i}$ and $\mathcal{A} \vdash^{\bullet} \lambda t_i.X_i : \tau_{X_i}$. $\mathcal{A}'$ contains assumptions over projection functions, and they do not appear in $\lambda t_i.X_i$, so by Theorem 1-*b)* we can add these assumptions and obtain $\mathcal{A} \oplus \mathcal{A}' \vdash^{\bullet} \lambda t_i.X_i : \tau_{X_i}$. We know that in $\mathcal{A}'$ there will appear an assumption $\{f_{X_i} : Gen(\tau_{X_i}, \mathcal{A})\}$ for the projection function of the variable X_i, with rule $f_{X_i}\ t_i \to X_i$. We know that $FTV(\mathcal{A}) = FTV(\mathcal{A} \oplus \mathcal{A}')$ because since all the assumptions in $\mathcal{A}$ are of the form $Gen(\tau_{X_i}, \mathcal{A})$ they will not add any type variable, and since no f_{X_i} appears in $\mathcal{A}$ they will not shadow any assumption. Then τ_{X_i} will be a variant of $Gen(\tau_{X_i}, \mathcal{A})$.

Therefore for every data variable X_i which is projected then $\mathcal{A} \vdash \lambda t_i.X_i : \tau_{X_i}$ and τ_{X_i} is a variant of $\mathcal{A} \oplus \mathcal{A}'(f_{X_i}) = Gen(\tau_{X_i}, \mathcal{A})$, so all the program rules $f_{X_i}\ t_i \to X_i \in \mathcal{P}'$ are well-typed wrt. $\mathcal{A} \oplus \mathcal{A}'$ and $wt_{\mathcal{A} \oplus \mathcal{A}'}(\mathcal{P}')$. □

The next theorem is one of the most important results of this work. It states that after a *let*-rewriting step using a well-typed program, the resulting expression has the same type as the original expression.

Theorem 3 (Subject Reduction).
If $\mathcal{A} \vdash e : \tau$ *and* $wt_{\mathcal{A}}(\mathcal{P})$ *and* $\mathcal{P} \vdash e \rightarrow^{l} e'$ *then* $\mathcal{A} \vdash e' : \tau$.

Proof. We proceed by case distinction over the rule of the *let*-rewriting relation $\rightarrow^{l}$ (Fig. 3.3) that we use to reduce e to e'.

(Fapp) If we reduce an expression e using the **(Fapp)** rule is because e has the form $f\ t_1\theta \ldots t_n\theta$ (being $f\ t_1 \ldots t_n \rightarrow r$ a rule in $\mathcal{P}$ and $\theta \in \mathcal{PS}ubst$) and e' is $r\theta$. In this case we want to prove that $\mathcal{A} \vdash r\theta : \tau$. Since $wt_{\mathcal{A}}(\mathcal{P})$, then $\mathcal{A} \vdash^{\bullet} \lambda t_1 \ldots \lambda t_n.r : \tau'_1 \rightarrow \ldots \rightarrow \tau'_n \rightarrow \tau'$, being $\tau'_1 \rightarrow \ldots \rightarrow \tau'_n \rightarrow \tau'$ a variant of $\mathcal{A}(f)$. We assume that the variables of the patterns $\overline{t_n}$ do not appear in $\mathcal{A}$ or in $Rng(\theta)$. The tree for this type derivation will be:

$$\dfrac{\mathcal{A}_1 \vdash t_1 : \tau'_1 \quad [\Lambda] \dfrac{\mathcal{A}_2 \vdash t_2 : \tau'_2 \quad [\Lambda] \dfrac{\begin{array}{cc}\mathcal{A}_n \vdash t_n : \tau'_n & \mathcal{A}_n \vdash r : \tau'\end{array}}{\vdots}}{\mathcal{A}_2 \vdash t_3 \ldots t_n : \tau'_3 \rightarrow \ldots \rightarrow \tau'_n \rightarrow \tau'}}{\mathcal{A}_1 \vdash \lambda t_2 \ldots t_n.r : \tau'_2 \rightarrow \ldots \rightarrow \tau'_n \rightarrow \tau'}\ [\Lambda]}{\mathcal{A} \vdash \lambda t_1 \ldots t_n.r : \tau'_1 \rightarrow \tau'_2 \ldots \rightarrow \tau'_n \rightarrow \tau'}\ [\Lambda]$$

where $\mathcal{A}_j \equiv (\ldots(\mathcal{A} \oplus \{\overline{X_{1m} : \tau''_{1m}}\}) \oplus \ldots) \oplus \{\overline{X_{jm} : \tau''_{jm}}\}$ and X_{ji} is the i-th variable of the pattern t_j. We can write $\mathcal{A}_n$ as $\mathcal{A} \oplus \mathcal{A}'$, being $\mathcal{A}'$ the set of assumption over the variables of the patterns. As these variables are all different (the left hand side of the rules is linear), by Theorem 1-b we can add the rest of the assumptions to the $\mathcal{A}_j$ to get $\mathcal{A}_n$ and the derivation will remain valid, so $\forall j \in [1, n].\ \mathcal{A}_n \vdash t_j : \tau'_j$. Besides $critVar_{\mathcal{A}}(\lambda t_1 \ldots \lambda t_n.r) = \emptyset$, so *a)* every variable X_{ji} which appears in r is transparent in the pattern t_j where it comes.

It is a premise that $\mathcal{A} \vdash f\ t_1\theta \ldots t_n\theta : \tau$, and the tree of the type derivation will be:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{A} \vdash f : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau}\ \textbf{[ID]} \quad \mathcal{A} \vdash t_1\theta : \tau_1}{\vdots}\ \textbf{[APP]}}{\mathcal{A} \vdash f\ t_1\theta \ldots t_{n-2}\theta : \tau_{n-1} \rightarrow (\tau_n \rightarrow \tau)}\ \textbf{[APP]} \quad \mathcal{A} \vdash t_{n-1}\theta : \tau_{n-1}}{\mathcal{A} \vdash f\ t_1\theta \ldots t_{n-1}\theta : \tau_n \rightarrow \tau}\ \textbf{[APP]} \quad \mathcal{A} \vdash t_n\theta : \tau_n}{\mathcal{A} \vdash f\ t_1\theta \ldots t_n\theta : \tau}\ \textbf{[APP]}$$

Because of that, we know that *b)* $\forall j \in [1,n].\ \mathcal{A} \vdash t_j\theta : \tau_j$ and $\mathcal{A} \vdash f : \tau_1 \to \ldots \to \tau_n \to \tau$, being $\tau_1 \to \ldots \to \tau_n \to \tau$ a generic instance of the type $\mathcal{A}(f)$. Then there will exists a type substitution π such that $(\tau'_1 \to \ldots \to \tau'_n \to \tau')\pi = \tau_1 \to \ldots \to \tau_n \to \tau$, so $\forall j \in [1,n].\ \tau'_j\pi = \tau_j$ and $\tau'\pi = \tau$. What is more, $Dom(\pi)$ does not contain any free type variable in $\mathcal{A}$, since π transforms a variant of the type of $\mathcal{A}(f)$ into a generic instance of the type of $\mathcal{A}(f)$. Then by Theorem 1-a $\mathcal{A}_n\pi \vdash t_j : \tau'_j\pi$, which is equal to *c)* $\mathcal{A} \oplus \mathcal{A}'\pi \vdash t_j : \tau'_j\pi$.

With *a)*, *b)* and *c)* and by Lemma 1 we can state that for every transparent variable X_{ji} in r then $\mathcal{A} \vdash X_{ji}\theta : \tau''_{ji}\pi$. None of the variables in $\mathcal{A}'$ appear in $X_{ji}\theta$, so by Theorem 1-b we can add these assumptions and obtain $\mathcal{A}_n \vdash X_{ji}\theta : \tau''_{ji}\pi$. According to the first derivation, we have $\mathcal{A}_n \vdash r : \tau'$. Here we can apply the Theorem 1-a again and get a derivation $\mathcal{A}_n\pi \vdash r : \tau'\pi$. Because $\mathcal{A}_n\pi \vdash X_{ji}\theta : \tau''_{ji}\pi$, then by Theorem 1-c $\mathcal{A}_n\pi \vdash r\theta : \tau'\pi$. As we have eliminated the variables in the expression, by Theorem 1-b we can delete their assumptions, obtaining a derivation $\mathcal{A}\pi \vdash r\theta : \tau'\pi$ (remember that $\mathcal{A}_n$ is $\mathcal{A} \oplus \mathcal{A}'$). And finally using the information we have about π, this derivation is equal to $\mathcal{A} \vdash r\theta : \tau$, the derivation we wanted to obtain.

(LetIn) In this case $\mathcal{A} \vdash e_1e_2 : \tau$ and $\mathcal{P} \vdash e_1e_2 \to^l let_m\ X = e_2\ in\ e_1$. The type derivation of e_1e_2 will have the form:

$$\textbf{[APP]}\frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \qquad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1e_2 : \tau}$$

With this information we could build a type judgment for the let$_m$ expression

$$\textbf{[LET}_m\textbf{]}\frac{\mathcal{A} \oplus \{X : \tau_1\} \vdash X : \tau_1 \qquad \mathcal{A} \vdash e_2 : \tau_1 \qquad \textbf{[APP]}\ \dfrac{\mathcal{A} \oplus \{X : \tau_1\} \vdash e_1 : \tau_1 \to \tau \qquad \mathcal{A} \oplus \{X : \tau_1\} \vdash X : \tau_1}{\mathcal{A} \oplus \{X : \tau_1\} \vdash e_1X : \tau}}{\mathcal{A} \vdash let_m\ X = e_2\ in\ e_1X : \tau}$$

$\mathcal{A} \oplus \{X : \tau_1\} \vdash X : \tau_1$ is a valid derivation because is an application of the [ID] rule. And since X is a fresh variable, by Theorem 1-b we can add the assumption and obtain $\mathcal{A} \oplus \{X : \tau_1\} \vdash e_1 : \tau_1 \to \tau$.

(Bind) We will distinguish between the let$_m$ and the let$_p$ case. In both cases we assume that the variable X is fresh.

let_m) In the let$_m$ case the type derivation will have the form:

$$\textbf{[LET}_m\textbf{]}\frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \qquad \mathcal{A} \vdash t : \tau_t \qquad \mathcal{A} \oplus \{X : \tau_t\} \vdash e : \tau}{\mathcal{A} \vdash let_m\ X = t\ in\ e : \tau}$$

As X is different from all the variables $\overline{X_n}$ of the pattern t, then by Theorem 1-b we can add the assumption over the variable X and obtain the derivation $\mathcal{A} \oplus \{X : \tau_t\} \vdash t : \tau_t$. Applying

the Theorem 1-c then $\mathcal{A} \oplus \{X : \tau_t\} \vdash e[X/t] : \tau$. X will not appear in $e[X/t]$, so again by Theorem 1-b we can eliminate the assumption, concluding that $\mathcal{A} \vdash e[X/t] : \tau$.

let_p) Here the type derivations will be:

$$\textbf{[LET}_p\textbf{]}\frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \qquad \mathcal{A} \vdash t : \tau_t \qquad \mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\} \vdash e : \tau}{\mathcal{A} \vdash let_p\ X = t\ in\ e : \tau}$$

and we want to prove that $\mathcal{A} \vdash e[X/t] : \tau$. We have a type derivation for $\mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\} \vdash e : \tau$, and according to Observation 7 there will be derivations $(\mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\}) \oplus \mathcal{A}'_i \vdash X : \tau_i$ for every appearance of X in e. In these cases, $\mathcal{A}'_i$ will only contain assumptions over variables $\overline{X_n}$ in let or lambda expressions of e. Suppose that all these variables have been renamed to fresh variables. We can create a type substitution π from the variables $\overline{\alpha_n}$ of τ_t which do not appear in $\mathcal{A}$ to fresh type variables $\overline{\beta_n}$. It is clear that $Gen(\tau_t, \mathcal{A})$ is equivalent to $Gen(\tau_t\pi, \mathcal{A})$, so $\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\} \vdash e : \tau$ is a valid derivation. By Theorem 1-a $\mathcal{A}\pi \vdash t : \tau_t\pi$, and since $\overline{\alpha_n}$ are not in $\mathcal{A}$ then $\mathcal{A} \vdash t : \tau_t\pi$. X and $\overline{X_n}$ are fresh so they do not appear in t and by Theorem 1-b we can add assumptions to the derivation $\mathcal{A} \vdash t : \tau_t\pi$, obtaining $(\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\}) \oplus \mathcal{A}'_i \vdash t : \tau_t\pi$. The types $\overline{\tau_n}$ will be generic instances of $Gen(\tau_t, \mathcal{A})$, and also of $Gen(\tau_t\pi, \mathcal{A})$. Then for each τ_i there will exist a type substitution π'_i from the generalized variables $\overline{\beta_n}$ in $Gen(\tau_t\pi, \mathcal{A})$ to types that will hold $\tau_t\pi\pi'_i \equiv \tau_i$. By Theorem 1-a we can convert $(\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\}) \oplus \mathcal{A}'_i \vdash t : \tau_t\pi$ into $((\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\}) \oplus \mathcal{A}'_i)\pi'_i \vdash t : \tau_t\pi\pi'_i$, and as $\overline{\beta_n}$ are fresh variables then $(\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\}) \oplus \mathcal{A}'_i \vdash t : \tau_t\pi\pi'_i$ (note that π'_i does not affect $Gen(\tau_t\pi, \mathcal{A})$ because the variables $\overline{\beta_n}$ are generalized). This way in every place of the original derivation where we have $(\mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\}) \oplus \mathcal{A}'_i \vdash X : \tau_i$ we could place a derivation $(\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\}) \oplus \mathcal{A}'_i \vdash t : \tau_i$. The resulting expression of this substitution will be $e[X/t]$, so $\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\} \vdash e[X/t] : \tau$. It is clear that X does not appear in $e[X/t]$, so by Theorem 1-b we can eliminate the assumption over the X and obtain a derivation $\mathcal{A} \vdash e[X/t] : \tau$, as we wanted to prove.

(Elim) In this case it does not matter what type of let expression it was (let$_m$ or let$_p$). The rewriting step will be of the form $\mathcal{P} \vdash let_*\ X = e_1\ in\ e_2 \rightarrow^l e_2$. The type derivation of $\mathcal{A} \vdash let_*\ X = e_1\ in\ e_2 : \tau$ will have a branch $\mathcal{A} \oplus \{X : \sigma'\} \vdash e_2 : \tau$ for some σ. Since we are using the **(Elim)** rule, X does not appear in e_2 so by Theorem 1-b we can derive the same type eliminating that assumption, obtaining $\mathcal{A} \vdash e_2 : \tau$.

(Flat$_m$) There are two cases, depending on the second let expression. In both cases we assume that $X \neq Y$.

- $\mathcal{P} \vdash let_m\ X = (let_m\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^l let_m\ Y = e_1\ in\ (let_m\ X = e_2\ in\ e_3)$.

The type derivation will be:

$$\textbf{[LET}_m\textbf{]}\frac{\mathcal{A} \oplus \{X : \tau_x\} \vdash X : \tau_x \qquad \textbf{[LET}_m\textbf{]}\ \dfrac{\begin{array}{c}\mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y \\ \mathcal{A} \vdash e_1 : \tau_y \\ \mathcal{A} \oplus \{Y : \tau_y\} \vdash e_2 : \tau_x\end{array}}{\mathcal{A} \vdash let_m\ Y = e_1\ in\ e_2 : \tau_x} \qquad \mathcal{A} \oplus \{X : \tau_x\} \vdash e_3 : \tau}{\mathcal{A} \vdash let_m\ X = (let_m\ Y = e_1\ in\ e_2)\ in\ e_3 : \tau}$$

Then we can build a type derivation

$$\textbf{[LET}_m\textbf{]}\frac{\mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y \qquad \mathcal{A} \vdash e_1 : \tau_y \qquad \textbf{[LET}_m\textbf{]}\ \dfrac{\begin{array}{c}(\mathcal{A} \oplus \{Y : \tau_y\}) \oplus \{X : \tau_x\} \vdash X : \tau_x \\ \mathcal{A} \oplus \{Y : \tau_y\} \vdash e_2 : \tau_x \\ (\mathcal{A} \oplus \{Y : \tau_y\}) \oplus \{X : \tau_x\} \vdash e_3 : \tau\end{array}}{\mathcal{A} \oplus \{Y : \tau_y\} \vdash let_m\ X = e_2\ in\ e_3 : \tau}}{\mathcal{A} \vdash let_m\ Y = e_1\ in\ (let_m\ X = e_2\ in\ e_3) : \tau}$$

The only two derivations which do not come from the hypotheses are $(\mathcal{A} \oplus \{Y : \tau_y\}) \oplus \{X : \tau_x\} \vdash X : \tau_x$ and $(\mathcal{A} \oplus \{Y : \tau_y\}) \oplus \{X : \tau_x\} \vdash e_3 : \tau$. The first is the application of the **[ID]** rule. From the hypotheses we have a derivation $\mathcal{A} \oplus \{X : \tau_x\} \vdash e_3 : \tau$. Since we are rewriting using the **(Flat)** rule, we are sure that Y is not in e_3 and by Theorem 1-b we can add the assumption over the Y, obtaining the derivation $(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{Y : \tau_y\} \vdash e_3 : \tau$. X is different from Y, so according to Observation 3 $(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{Y : \tau_y\}$ is the same as $(\mathcal{A} \oplus \{Y : \tau_y\}) \oplus \{X : \tau_x\}$. Therefore $(\mathcal{A} \oplus \{Y : \tau_y\}) \oplus \{X : \tau_x\} \vdash e_3 : \tau$ is a valid derivation.

- $\mathcal{P} \vdash let_m\ X = (let_p\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^l let_p\ Y = e_1\ in\ (let_m\ X = e_2\ in\ e_3)$. Similar to the previous case.

(Flat$_p$) We will treat the two different cases:

- $\mathcal{P} \vdash let_p\ X = (let_p\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^l let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3)$.

  The type derivation of the original expression is (being $\mathcal{A}_Y \equiv \mathcal{A} \oplus \{Y : Gen(\tau_y, \mathcal{A})\}$)

$$\textbf{[LET}_p\textbf{]}\frac{\mathcal{A} \oplus \{X : \tau_x\} \vdash X : \tau_x \qquad \textbf{[LET}_p\textbf{]}\ \dfrac{\begin{array}{c}\mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y \\ \mathcal{A} \vdash e_1 : \tau_y \\ \mathcal{A}_Y \vdash e_2 : \tau_x\end{array}}{\mathcal{A} \vdash let_p\ Y = e_1\ in\ e_2 : \tau_x} \qquad \mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_3 : \tau}{\mathcal{A} \vdash let_p\ X = (let_p\ Y = e_1\ in\ e_2)\ in\ e_3 : \tau}$$

With these derivations as hypothesis we can build a type derivation of the new expression

$$\textbf{[LET}_p\textbf{]}\frac{\mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y \quad \mathcal{A} \vdash e_1 : \tau_y \quad \textbf{[LET}_p\textbf{]}\ \dfrac{\begin{array}{c}\mathcal{A}_Y \oplus \{X : \tau_x\} \vdash X : \tau_x \\ \mathcal{A}_Y \vdash e_2 : \tau_x \\ \mathcal{A}_Y \oplus \{X : Gen(\tau_x, \mathcal{A}_Y)\} \vdash e_3 : \tau\end{array}}{\mathcal{A}_Y \vdash let_p\ X = e_2\ in\ e_3 : \tau}}{\mathcal{A} \vdash let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3) : \tau}$$

$\mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y$, $\mathcal{A} \vdash e_1 : \tau_y$ and $\mathcal{A}_Y \vdash e_2 : \tau_x$ are the same derivations that appear in the original type derivation; and $\mathcal{A}_Y \oplus \{X : \tau_x\} \vdash X : \tau_x$ holds trivially applying the **[ID]** rule. But the derivation $\mathcal{A}_Y \oplus \{X : Gen(\tau_x, \mathcal{A}_Y)\} \vdash e_3 : \tau$ has to be proven. As before, since $Y \notin FV(e_3)$ by Theorem 1-b we can add an assumption over the Y and the derivation $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\}) \oplus \{Y : Gen(\tau_y, \mathcal{A})\} \vdash e_3 : \tau$ will remain valid. Because $X \neq Y$ then by Observation 3 $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\}) \oplus \{Y : Gen(\tau_y, \mathcal{A})\}$ is the same as $(\mathcal{A} \oplus \{Y : Gen(\tau_y, \mathcal{A})\}) \oplus \{X : Gen(\tau_x, \mathcal{A})\}$, and the derivation $(\mathcal{A} \oplus \{Y : Gen(\tau_y, \mathcal{A})\}) \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_3 : \tau$ will be correct. Clearly $Gen(\tau_x, \mathcal{A}_Y)$ is not equal to $Gen(\tau_x, \mathcal{A})$ because a previous assumption for Y can be shadowed so that some free type variables in $\mathcal{A}$ are not in $\mathcal{A}_Y$. In the generalization step this means that some variables can be generalized in $Gen(\tau_x, \mathcal{A}_Y)$ but not in $Gen(\tau_x, \mathcal{A})$. The other case never happens because adding $\{Y : Gen(\tau_y, \mathcal{A})\}$ to $\mathcal{A}$ never adds free type variables: if some type variable in τ_y is not in $FTV(\mathcal{A})$ then it will be generalized and will not be in $FTV(\mathcal{A}_Y)$ either. Therefore $Gen(\tau_x, \mathcal{A}_Y) \succ Gen(\tau_x, \mathcal{A})$, and by Theorem 1-d the derivation $(\mathcal{A} \oplus \{Y : Gen(\tau_y, \mathcal{A})\}) \oplus \{X : Gen(\tau_x, \mathcal{A}_Y)\} \vdash e_3 : \tau$ is valid.

– $\mathcal{P} \vdash let_p\ X = (let_m\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^l let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3)$.

The type derivation of the original expression is:

$$\textbf{[LET}_p\textbf{]}\frac{\mathcal{A} \oplus \{X : \tau_x\} \vdash X : \tau_x \quad \textbf{[LET}_m\textbf{]}\ \dfrac{\begin{array}{c}\mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y \\ \mathcal{A} \vdash e_1 : \tau_y \\ \mathcal{A} \oplus \{Y : \tau_y\} \vdash e_2 : \tau_x\end{array}}{\mathcal{A} \vdash let_m\ Y = e_1\ in\ e_2 : \tau_x} \quad \mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_3 : \tau}{\mathcal{A} \vdash let_p\ X = (let_m\ Y = e_1\ in\ e_2)\ in\ e_3 : \tau}$$

and we want to build one of the form (being $\mathcal{A}_Y \equiv \mathcal{A} \oplus \{Y : Gen(\tau_y, \mathcal{A})\}$):

$$\textbf{[LET}_p\textbf{]}\frac{\mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y \quad \mathcal{A} \vdash e_1 : \tau_y \quad \textbf{[LET}_p\textbf{]}\ \dfrac{\begin{array}{c}\mathcal{A}_Y \oplus \{X : \tau_x\} \vdash X : \tau_x \\ \mathcal{A}_Y \vdash e_2 : \tau_x \\ \mathcal{A}_Y \oplus \{X : Gen(\tau_x, \mathcal{A}_Y)\} \vdash e_3 : \tau\end{array}}{\mathcal{A}_Y \vdash let_p\ X = e_2\ in\ e_3 : \tau}}{\mathcal{A} \vdash let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3) : \tau}$$

The derivations $\mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y$ and $\mathcal{A} \vdash e_1 : \tau_y$ come from the original derivation; and $\mathcal{A}_Y \oplus \{X : \tau_x\} \vdash X : \tau_x$ is the trivial application of the [ID] rule. From the original derivation we have $\mathcal{A} \oplus \{Y : \tau_y\} \vdash e_2 : \tau_x$. It is easy to see that $Gen(\tau_y, \mathcal{A}) \succ \tau_y$, so by Theorem 1-d $\mathcal{A}_Y \vdash e_2 : \tau_x$. We also have from the original derivation that $\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_3 : \tau$. We know that $Y \notin FV(e_3)$, so by Theorem 1-b we can add an assumption over that variable and the derivation $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\}) \oplus \{Y : Gen(\tau_y, \mathcal{A})\} \vdash e_3 : \tau$ will be valid. X is different from Y, so according to Observation 3 the set of assumptions $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\}) \oplus \{Y : Gen(\tau_y, \mathcal{A})\}$ is the same as $(\mathcal{A} \oplus \{Y : Gen(\tau_y, \mathcal{A})\}) \oplus \{X : Gen(\tau_x, \mathcal{A})\}$. By the same reasons given in the previous case $Gen(\tau_x, \mathcal{A}_Y) \succ Gen(\tau_x, \mathcal{A})$, so by Theorem 1-d the derivation $\mathcal{A}_Y \oplus \{X : Gen(\tau_x, \mathcal{A}_Y)\} \vdash e_3 : \tau$ will be valid.

(LetAp) We will distinguish between the different let expressions.

let_m) The rewriting step is $\mathcal{P} \vdash (let_m\ X = e_1\ in\ e_2)e_3 \rightarrow^l let_m\ X = e_1\ in\ e_2e_3$. The type derivation of $(let_m\ X = e_1\ in\ e_2)e_3$ is:

$$\textbf{[APP]}\ \frac{\textbf{[LET}_m\textbf{]}\ \dfrac{\begin{array}{c}\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \\ \mathcal{A} \vdash e_1 : \tau_t \\ \mathcal{A} \oplus \{X : \tau_t\} \vdash e_2 : \tau_1 \rightarrow \tau\end{array}}{\mathcal{A} \vdash let_m\ X = e_1\ in\ e_2 : \tau_1 \rightarrow \tau} \qquad \mathcal{A} \vdash e_3 : \tau_1}{\mathcal{A} \vdash (let_m\ X = e_1\ in\ e_2)e_3 : \tau}$$

We want to construct a type derivation of the form:

$$\textbf{[LET}_m\textbf{]}\ \frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \qquad \mathcal{A} \vdash e_1 : \tau_t \qquad \textbf{[APP]}\ \dfrac{\begin{array}{c}\mathcal{A} \oplus \{X : \tau_t\} \vdash e_2 : \tau_1 \rightarrow \tau \\ \mathcal{A} \oplus \{X : \tau_t\} \vdash e_3 : \tau_1\end{array}}{\mathcal{A} \oplus \{X : \tau_t\} \vdash e_2e_3 : \tau}}{\mathcal{A} \vdash let_m\ X = e_1\ in\ e_2e_3 : \tau}$$

All the derivations appear in the original derivation, except $\mathcal{A} \oplus \{X : \tau_t\} \vdash e_3 : \tau_1$. Because we are using **(LetAp)**, we are sure that X does not appear in $FV(e_3)$. From the original derivation we have that $\mathcal{A} \vdash e_3 : \tau_1$, and by Theorem 1-b we can add an assumption over the variable X and obtain the derivation $\mathcal{A} \oplus \{X : \tau_t\} \vdash e_3 : \tau_1$.

let_p) Similar to the let_m) case.

(Contx) We have a derivation $\mathcal{A} \vdash \mathcal{C}[e] : \tau$, so according to the Observation 7 in that derivation will appear a derivation a) $\mathcal{A} \oplus \mathcal{A}' \vdash e : \tau'$, being $\mathcal{A}'$ a set of assumptions over variables. If we apply the

rule **(Contx)** to reduce an expression $\mathcal{C}[e]$ is because we reduce the expression e using any of the other rules of the *let*-rewriting relation b) $\mathcal{P} \vdash e \rightarrow^{l} e'$. We also know by Observation 8 that c) $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$. With a), b) and c) the Induction Hypothesis states that $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau'$, and by Lemma 6 then $\mathcal{A} \vdash \mathcal{C}[e'] : \tau$.

□

The next theorem states that $\Vdash$ (resp. $\Vdash^{\bullet}$) is sound wrt. $\vdash$ (resp. $\vdash^{\bullet}$), i.e., that the type found by the inference is a valid type for the expression applying the substitution to the set of assumptions.

Theorem 4 (Soundness of $\Vdash^{?}$)
1) $\mathcal{A} \Vdash e : \tau|\pi \Longrightarrow \mathcal{A}\pi \vdash e : \tau$

Proof.
We proceed by induction over the size of the type inference $\mathcal{A} \Vdash e : \tau|\pi$.
Base Case

[iID] We have a type inference of the form:

$$\textbf{[iID]}\frac{}{\mathcal{A} \Vdash g : \tau|id}$$

where $\mathcal{A}(g) = \sigma$ and τ is a variant of σ. It is clear that if τ is a variant of σ it is also a generic instance of σ, and $\mathcal{A}\ id \equiv \mathcal{A}$ so the following type derivation is valid:

$$\textbf{[ID]}\frac{}{\mathcal{A} \vdash g : \tau}$$

Induction Step

[iAPP] The type inference is:

$$\textbf{[iAPP]}\frac{\mathcal{A} \Vdash e_1 : \tau_1|\pi_1 \qquad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2|\pi_2}{\mathcal{A} \Vdash e_1 e_2 : \alpha\pi|\pi_1\pi_2\pi}$$

where $\pi = mgu(\tau_1\pi_2, \tau_2 \rightarrow \alpha)$, being α a fresh type variable. By the Induction Hypothesis we have that $\mathcal{A}\pi_1 \vdash e_1 : \tau_1$ and $\mathcal{A}\pi_1\pi_2 \vdash e_2 : \tau_2$. We can apply Theorem 1-a to both derivations and obtain $\mathcal{A}\pi_1\pi_2\pi \vdash e_1 : \tau_1\pi_2\pi$ and $\mathcal{A}\pi_1\pi_2\pi \vdash e_2 : \tau_2\pi$. Since we know that $\tau_1\pi_2\pi = (\tau_2 \rightarrow \alpha)\pi = \tau_2\pi \rightarrow \alpha\pi$ then we can construct the type derivation:

$$\textbf{[APP]}\frac{\mathcal{A}\pi_1\pi_2\pi \vdash e_1 : \tau_2\pi \to \alpha\pi \qquad \mathcal{A}\pi_1\pi_2\pi \vdash e_2 : \tau_2\pi}{\mathcal{A}\pi_1\pi_2\pi \vdash e_1 e_2 : \alpha\pi}$$

[iΛ] The type inference will be of the form:

$$\textbf{[i}\Lambda\textbf{]}\frac{\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t \qquad (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t \Vdash e : \tau|\pi}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi \to \tau|\pi_t\pi}$$

where $\overline{\alpha_n}$ are fresh type variables. By the Induction Hypothesis we have that $\mathcal{A}\pi_t \oplus \{\overline{X_n : \alpha_n\pi_t}\} \vdash t : \tau_t$ and $\mathcal{A}\pi_t\pi \oplus \{\overline{X_n : \alpha_n\pi_t\pi}\} \vdash e : \tau$. We can apply Theorem 1-a to the first derivation and obtain $\mathcal{A}\pi_t\pi \oplus \{\overline{X_n : \alpha_n\pi_t\pi}\} \vdash t : \tau_t\pi$. Therefore the following type derivation is correct:

$$\textbf{[}\Lambda\textbf{]}\frac{\mathcal{A}\pi_t\pi \oplus \{\overline{X_n : \alpha_n\pi_t\pi}\} \vdash t : \tau_t\pi \qquad \mathcal{A}\pi_t\pi \oplus \{\overline{X_n : \alpha_n\pi_t\pi}\} \vdash e : \tau}{\mathcal{A}\pi_t\pi \vdash \lambda t.e : \tau_t\pi \to \tau}$$

[iLET$_m$] In this case the type inference will be:

$$\textbf{[iLET}_m\textbf{]}\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t \\ \mathcal{A}\pi_t \Vdash e : \tau_1|\pi_1 \\ (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2|\pi_2\end{array}}{\mathcal{A} \Vdash let_m\ t = e_1\ in\ e_2 : \tau_2|\pi_t\pi_1\pi\pi_2}$$

where $\overline{\alpha_n}$ are fresh type variables and $\pi = mgu(\tau_t\pi_1, \tau_1)$. By the Induction Hypothesis we have that $\mathcal{A}\pi_t \oplus \{\overline{X_n : \alpha_n\pi_t}\} \vdash t : \tau_t$, $\mathcal{A}\pi_t\pi_1 \vdash e : \tau_1$ and $\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_n : \alpha_n\pi_t\pi_1\pi\pi_2}\} \vdash e_2 : \tau_2$. We can apply Theorem 1-a to the first two derivations and obtain $\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_n : \alpha_n\pi_t\pi_1\pi\pi_2}\} \vdash t : \tau_t\pi_1\pi\pi_2$ and $\mathcal{A}\pi_t\pi_1\pi\pi_2 \vdash e : \tau_1\pi\pi_2$. Finally, as $\tau_t\pi_1\pi = \tau_1\pi$ then we can build a type derivation of the form:

$$\textbf{[LET}_m\textbf{]}\frac{\begin{array}{c}\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_n : \alpha_n\pi_t\pi_1\pi\pi_2}\} \vdash t : \tau_1\pi\pi_2 \\ \mathcal{A}\pi_t\pi_1\pi\pi_2 \vdash e : \tau_1\pi\pi_2 \\ \mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_n : \alpha_n\pi_t\pi_1\pi\pi_2}\} \vdash e_2 : \tau_2\end{array}}{\mathcal{A}\pi_t\pi_1\pi\pi_2 \vdash let_m\ t = e_1\ in\ e_2 : \tau_2}$$

[iLET$^X_{pm}$] The inference will be:

$$\textbf{[LET}^X_{pm}\textbf{]}\frac{\mathcal{A} \Vdash e_1 : \tau_1|\pi_1 \qquad \mathcal{A}\pi_1 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\} \Vdash e_2 : \tau_2|\pi_2}{\mathcal{A} \Vdash let_{pm}\ X = e_1\ in\ e_2 : \tau_2|\pi_1\pi_2}$$

By the Induction Hypothesis we have the type derivations $\mathcal{A}\pi_1 \vdash e_1 : \tau_1$ and $\mathcal{A}\pi_1\pi_2 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\pi_2\} \vdash e_2 : \tau_2$. We can construct a type substitution $\pi \in \mathcal{TS}ubst$ such that maps the type variables in $FTV(\tau_1) \setminus FTV(\mathcal{A}\pi_1)$ to fresh variables. Then it is clear that $Gen(\tau_1, \mathcal{A}\pi_1) = Gen(\tau_1\pi, \mathcal{A}\pi_1)$. On the other hand, all the variables in $\tau_1\pi$ which are not in $FTV(\mathcal{A}\pi_1)$ are fresh so they do not appear in π_2, and by Lemma 8 $Gen(\tau_1\pi, \mathcal{A}\pi_1)\pi_2 = Gen(\tau_1\pi\pi_2, \mathcal{A}\pi_1\pi_2)$.

Therefore the type derivation $\mathcal{A}\pi_1\pi_2 \oplus \{X : Gen(\tau_1\pi\pi_2, \mathcal{A}\pi_1\pi_2)\} \vdash e_2 : \tau_2$ is correct. By Theorem 1-a we obtain $\mathcal{A}\pi_1\pi\pi_2 \vdash e_1 : \tau_1\pi\pi_2$, and as $Dom(\pi) \cap FTV(\mathcal{A}\pi_1) = \emptyset$ then $\mathcal{A}\pi_1\pi_2 \vdash e_1 : \tau_1\pi\pi_2$.

Finally with these derivations we can build the type derivation we intended:

$$\textbf{[LET}^{X}_{pm}\textbf{]}\frac{\mathcal{A}\pi_1\pi_2 \vdash e_1 : \tau_1\pi\pi_2 \qquad \mathcal{A}\pi_1\pi_2 \oplus \{X : Gen(\tau_1\pi\pi_2, \mathcal{A}\pi_1\pi_2)\} \vdash e_2 : \tau_2}{\mathcal{A}\pi_1\pi_2 \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau_2}$$

[iLET$^{h}_{pm}$] This case is similar to the **[LET$_m$]** case.

[iLET$_p$] In this case we have an inference of the form:

$$\textbf{[iLET}_p\textbf{]}\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t \\ \mathcal{A}\pi_t \Vdash e_1 : \tau_1|\pi_1 \\ \mathcal{A}\pi_t\pi_1\pi \oplus \{\overline{X_n : Gen(\alpha_n\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi)}\} \Vdash e_2 : \tau_2|\pi_2\end{array}}{\mathcal{A} \Vdash let_p\ t = e_1\ in\ e_2 : \tau_2|\pi_t\pi_1\pi\pi_2}$$

where $\pi = mgu(\tau_t\pi_1, \tau_1)$. By the Induction Hypothesis we have that $\mathcal{A}\pi_t \oplus \{\overline{X_n : \alpha_n\pi_t}\} \vdash t : \tau_t$, $\mathcal{A}\pi_t\pi_1 \vdash e_1 : \tau_1$ and $\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_n : Gen(\alpha_n\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi)\pi_2}\} \vdash e_2 : \tau_2$. Let be $\overline{\beta_m}$ the type variables in all the types $\overline{\alpha_n\pi_t\pi_1\pi}$ which do not appear in $\mathcal{A}\pi_t\pi_1\pi$. We can create a type substitution π' from $\overline{\beta_m}$ to fresh variables. It is clear that $Gen(\alpha_i\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi) = Gen(\alpha_i\pi_t\pi_1\pi\pi', \mathcal{A}\pi_t\pi_1\pi)$, as π' only substitutes the variables that will be generalized by fresh ones which will also be generalized, so it is a renaming of the bounded variables (Observation 1). Therefore the derivation $\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_n : Gen(\alpha_n\pi_t\pi_1\pi\pi', \mathcal{A}\pi_t\pi_1\pi)\pi_2}\} \vdash e_2 : \tau_2$ is also valid. Applying the Theorem 1-a to the first two derivations we obtain $\mathcal{A}\pi_t\pi_1\pi\pi'\pi_2 \oplus \{\overline{X_n : \alpha_n\pi_t\pi_1\pi\pi'\pi_2}\} \vdash t : \tau_t\pi_1\pi\pi'\pi_2$ and $\mathcal{A}\pi_t\pi_1\pi\pi'\pi_2 \vdash e_1 : \tau_1\pi\pi'\pi_2$. By construction, no variable in $Dom(\pi')$ or $Rng(\pi')$ is in $FTV(\mathcal{A}\pi_t\pi_1\pi)$, so $\mathcal{A}\pi_t\pi_1\pi\pi'\pi_2 = \mathcal{A}\pi_t\pi_1\pi\pi_2$. By Lemma 8 $Gen(\alpha_i\pi_t\pi_1\pi\pi', \mathcal{A}\pi_t\pi_1\pi)\pi_2 = Gen(\alpha_i\pi_t\pi_1\pi\pi'\pi_2, \mathcal{A}\pi_t\pi_1\pi\pi_2)$, so the derivation $\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_n : Gen(\alpha_n\pi_t\pi_1\pi\pi'\pi_2, \mathcal{A}\pi_t\pi_1\pi\pi_2)}\} \vdash e_2 : \tau_2$ is correct.

With these derivations as premises we can build the expected one:

$$\textbf{[LET}_p\textbf{]}\frac{\begin{array}{c}\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_n : \alpha_n\pi_t\pi_1\pi\pi'\pi_2}\} \vdash t : \tau_1\pi\pi'\pi_2 \\ \mathcal{A}\pi_t\pi_1\pi\pi_2 \vdash e_1 : \tau_1\pi\pi'\pi_2 \\ \mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_n : Gen(\alpha_n\pi_t\pi_1\pi\pi'\pi_2, \mathcal{A}\pi_t\pi_1\pi\pi_2)}\} \vdash e_2 : \tau_2\end{array}}{\mathcal{A}\pi_t\pi_1\pi\pi_2 \vdash let_p\ t = e_1\ in\ e_2 : \tau_2}$$

(remembering that $\tau_t\pi_1\pi = \tau_1\pi$ because of π is a mgu).

2) $\mathcal{A} \Vdash^{\bullet} e : \tau|\pi \Longrightarrow \mathcal{A}\pi \vdash^{\bullet} e : \tau$

By definition of $\Vdash^{\bullet}$ we have that $\mathcal{A} \Vdash e : \tau$ and $critVar_{\mathcal{A}\pi}(e)$. Applying the soundness of $\Vdash$ (Theorem 4) we have that $\mathcal{A}\pi \vdash e : \tau$. Since $\mathcal{A}\pi \vdash e : \tau$ and $critVar_{\mathcal{A}\pi}(e)$, then by definition of $\vdash^{\bullet}$ we have $\mathcal{A}\pi \vdash^{\bullet} e : \tau$. □

The theorem that follows states that if an expression e admits any type τ applying a substitution π' to the set of assumptions, then the inference $\Vdash$ will find a type and a substitution that are more general than τ and π'.

Theorem 5 (Completeness of $\Vdash$ wrt $\vdash$).
$\mathcal{A}\pi' \vdash e : \tau' \Longrightarrow \exists \tau, \pi, \pi''.\ \mathcal{A} \Vdash e : \tau|\pi \wedge \mathcal{A}\pi\pi'' = \mathcal{A}\pi' \wedge \tau\pi'' = \tau'$.

Proof.
This proof has similarities with the proof of completeness of algorithm $\mathscr{W}$ in [19]. We proceed by induction over the size of the type derivation.
Base Case

[ID] In this case we have a type derivation:

$$\textbf{[ID]}\frac{}{\mathcal{A}\pi' \vdash s : \tau'}$$

if $\mathcal{A}\pi'(s) = \sigma$ and $\sigma \succ \tau'$. Let's suppose that $\mathcal{A}(s) = \forall\overline{\alpha_n}.\tau''$ (with $\overline{\alpha}$ fresh variables), then $\sigma \equiv (\forall\overline{\alpha_n}.\tau'')\pi' = \forall\overline{\alpha_n}.(\tau''\pi')$. Since $\sigma \succ \tau'$ then there exists a type substitution $[\overline{\alpha_n/\tau_n}]$ such that $\tau' = (\tau''\pi')[\overline{\alpha_n/\tau_n}]$.

Let $\overline{\beta_n}$ be fresh variables. As $\tau''[\overline{\alpha_n/\beta_n}]$ is a variant of $\forall\overline{\alpha_n}.\tau''$ then the following type inference is correct:

$$\textbf{[iID]}\frac{}{\mathcal{A} \Vdash s : \tau''[\overline{\alpha_n/\beta_n}]|id}$$

There is also a type substitution $\pi'' \equiv \pi'[\overline{\beta_n/\tau_n}]$ such that $\tau''[\overline{\alpha_n/\beta_n}]\pi'' = \tau''[\overline{\alpha_n/\beta_n}]\pi'[\overline{\beta_n/\tau_n}] = (\tau''\pi')[\overline{\alpha_n/\beta_n}][\overline{\beta_n/\tau_n}] = (\tau''\pi')[\overline{\alpha_n/\tau_n}] = \tau'$. Finally, it is clear that $\mathcal{A}id\pi'' = \mathcal{A}id\pi'[\overline{\beta_n/\tau_n}] = \mathcal{A}\pi'[\overline{\beta_n/\tau_n}] = \mathcal{A}\pi'$ because $\overline{\beta_n}$ are fresh and cannot occur in $FTV(\mathcal{A}\pi')$.

Induction Step

[APP] The type derivation will be:

$$\textbf{[APP]}\frac{\mathcal{A}\pi' \vdash e_1 : \tau_1' \to \tau' \quad \mathcal{A}\pi' \vdash e_2 : \tau_1'}{\mathcal{A}\pi' \vdash e_1 e_2 : \tau'}$$

By the Induction Hypothesis we know that $\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1$ and there is a type substitution π_1'' such that $\tau_1\pi_1'' = \tau_1' \to \tau'$ and $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi_1''$. Since $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi_1''$ then the derivation $(\mathcal{A}\pi_1)\pi_1'' \vdash e_2 : \tau_1'$ is correct, and again by the Induction Hypothesis we know that $\mathcal{A}\pi_1 \Vdash e_2 : \tau_2 | \pi_2$ and that there exists a type substitution π_2'' such that $\tau_2\pi_2'' = \tau_1'$ and $\mathcal{A}\pi_1\pi_1'' = \mathcal{A}\pi_1\pi_2\pi_2''$. We can assume that π_2'' is minimal, so $Dom(\pi_2'') \subseteq FTV(\tau_2) \cup FTV(\mathcal{A}\pi_1\pi_2)$.

In order to prove that the existence of a type inference $\mathcal{A} \Vdash e_1 : \alpha\pi | \pi_1\pi_2\pi$ we need to prove that there exists a most general unifier for $\tau_1\pi_2$ and $\tau_2 \to \alpha$ (being α a fresh variable). For that, we will construct a type substitution π_u which will unify these two types. We know that $\mathcal{A}\pi_1\pi_1'' = \mathcal{A}\pi_1\pi_2\pi_2''$, so for all the variables which are free in $\mathcal{A}\pi_1$ then $\pi_1'' = \pi_2\pi_2''$. Let α a fresh type variable, $B = Dom(\pi_1'') \setminus FTV(\mathcal{A}\pi_1)$ and $\pi_u \equiv \pi_2'' + \pi_1''|_B + [\alpha/\tau']$. π_u is well defined because the domains of the three substitutions are disjoints. According to Observation 5, the variables in $FTV(\tau_2)$, $Dom(\pi_2)$ or $Rng(\pi_2)$ which are not in $FTV(\mathcal{A}\pi_1)$ are fresh variables and cannot occur in B. Since the variables in B are neither in $FTV(\mathcal{A}\pi_1)$ nor in $Rng(\pi_2)$ then they do not appear in $FTV(\mathcal{A}\pi_1\pi_2)$ either; and as π_2'' is minimal then no variable in B could occur in $Dom(\pi_2'')$. Besides α is fresh, and it can occur neither in π_2'' nor in $\pi_1''|_B$. Applying π_u to $\tau_2 \to \alpha$ we obtain $(\tau_2 \to \alpha)\pi_u = \tau_2\pi_u \to \alpha\pi_u = \tau_2\pi_2'' \to \alpha[\alpha/\tau'] = \tau_1' \to \tau'$. On the other hand, $\tau_1\pi_2\pi_u = \tau_1' \to \tau'$ because if a type variable of τ_1 is in $\mathcal{A}\pi_1$ then $\tau_1\pi_2\pi_u = \tau_1\pi_2\pi_2'' = \tau_1\pi_1'' = \tau_1' \to \tau'$, and if not it will be in B and π_2 will not affect it, so $\tau_1\pi_2\pi_u = \tau_1\pi_u = \tau_1\pi_1''|_B = \tau_1' \to \tau'$. Since π_u is an unifier, then there will exists a most general unifier π of $\tau_1\pi_2$ and $\tau_2 \to \alpha$ [63]. Therefore the following type inference is correct:

$$\textbf{[iAPP]}\frac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \vdash e_1 e_2 : \alpha\pi | \pi_1\pi_2\pi}$$

Now we have to prove that there exists a type substitution π'' such that $\alpha\pi\pi'' = \tau'$ and $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi_2\pi\pi''$. This is easy defining π'' such that $\pi_u = \pi\pi''$ (which is well defined as π_u is an unifier and π is the most general unifier). Then it is clear that $\alpha\pi\pi'' = \alpha\pi_u = \alpha[\alpha/\tau'] = \tau'$ and $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi_1'' = \mathcal{A}\pi_1\pi_2\pi_2'' = \mathcal{A}\pi_1\pi_2\pi_u = \mathcal{A}\pi_2\pi_2\pi\pi''$.

[Λ] We assume that the variables $\overline{X_n}$ in the pattern t do not appear in $\mathcal{A}\pi'$ (nor in $\mathcal{A}$). In this case the type derivation is:

$$\textbf{[}\Lambda\textbf{]}\frac{\mathcal{A}\pi' \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t' \quad \mathcal{A}\pi' \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau'}{\mathcal{A}\pi' \vdash \lambda t.e : \tau_t' \to \tau'}$$

Let $\overline{\alpha_n}$ be fresh type variables and $\pi_g \equiv [\overline{\alpha_n/\tau_n}]$. Then the first derivation is equal to $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi'\pi_g \vdash t : \tau'_t$. By the Induction Hypothesis we know that $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t$ and that exists a type substitution π''_t such that $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi'\pi_g = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi''_t$ and $\tau_t\pi''_t = \tau'_t$. Because the data variables $\overline{X_n}$ do not appear in $\mathcal{A}$, then it is true that $\mathcal{A}\pi'\pi_g = \mathcal{A}\pi' = \mathcal{A}\pi_t\pi''_t$ and for every type variable $\alpha_i\pi'\pi_g = \alpha_i\pi_g = \tau_i = \alpha\pi_t\pi''_t$.

Using these equalities we can write $\mathcal{A}\pi' \oplus \{\overline{X_n : \tau_n}\}$ as $\mathcal{A}\pi_t\pi''_t \oplus \{\overline{X_n : \alpha_n\pi_t\pi''_t}\}$, that is the same as $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi''_t$. Then, the second derivation is equal to $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi''_t \vdash e : \tau'$, and by the Induction Hypothesis $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t \Vdash e : \tau_e|\pi_e$ and there exists a type substitution π''_e such that $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi''_t = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_e\pi''_e$ and $\tau_e\pi''_e = \tau'$. As before, it is also true that $\mathcal{A}\pi_t\pi''_t = \mathcal{A}\pi_t\pi_e\pi''_e$ and for every type variable $\alpha_i\pi_t\pi''_t = \alpha\pi_t\pi_e\pi''_e$. We can assume that π''_e is minimal, so $Dom(\pi''_e) \subseteq FTV(\tau_e) \cup FTV((\mathcal{A} \cup \{\overline{X_n : \alpha_n}\})\pi_t\pi_e)$. Therefore the type inference for the lambda expression exists and have the form:

$$\textbf{[i}\Lambda\textbf{]}\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t \\ (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t \Vdash e : \tau_e|\pi_e\end{array}}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi_e \to \tau_e|\pi_t\pi_e}$$

Now we have to prove that there exists a type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi_t\pi_e\pi''$ and $(\tau_t\pi_e \to \tau_e)\pi'' = \tau'_t \to \tau'$. Let be $B \equiv Dom(\pi''_t) \setminus FTV((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$ and $\pi'' \equiv \pi''_t|_B + \pi''_e$, which is well defined because the domains are disjoints. According to Observation 5, the variables which are not in $FTV((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$ and appear in $FTV(\tau_e)$, $Dom(\pi_e)$ or in $Rng(\pi_e)$ are fresh, so they cannot be in B. As these variables do not appear in $Rng(\pi_e)$ then they do not appear in $FTV((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_e)$; so the variables in B are not in $Dom(\pi''_e)$ and the domains of π''_e and $\pi''_t|_B$ are disjoints.

It is clear that $\mathcal{A}\pi' = \mathcal{A}\pi_t\pi''_t = \mathcal{A}\pi_t\pi_e\pi''_e = \mathcal{A}\pi_t\pi_e\pi''$ because π''_e is part of π''. To prove that $(\tau_t\pi_e \to \tau_e)\pi'' = \tau'_t \to \tau'$ we need to prove that $\tau_t\pi_e\pi'' = \tau'_t$ and $\tau_e\pi'' = \tau'$. The second part is straightforward because $\tau' = \tau_e\pi''_e = \tau_e\pi''$. To prove the first one we will distinguish over the type variables in τ_t. For all the type variables of τ_t which are in $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t$ (i.e. they are not in B) we know that $\tau_t\pi_e\pi'' = \tau_t\pi_e\pi''_e = \tau_t\pi''_t = \tau'_t$ because $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi''_t = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_e\pi''_e$. For the variables in τ_t which are in B the case is simpler because we know they do not appear in $Dom(\pi_e)$, therefore so $\tau_t\pi_e\pi'' = \tau_t\pi'' = \tau_t\pi''_t|_B = \tau'_t$.

[LET$_m$] We assume that the variables $\overline{X_n}$ of the pattern t are fresh and do not occur in $\mathcal{A}\pi'$ (nor in $\mathcal{A}$). Then the type derivation will be:

$$\textbf{[LET}_m\textbf{]}\frac{\begin{array}{c}\mathcal{A}\pi' \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau'_t \\ \mathcal{A}\pi' \vdash e_1 : \tau'_t \\ \mathcal{A}\pi' \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau'\end{array}}{\mathcal{A}\pi' \vdash let_m\ t = e_1\ in\ e_2 : \tau'}$$

Let $\overline{\alpha_n}$ be fresh type variables, and $\pi_g \equiv [\overline{\alpha_n/\tau_n}]$. Since $\overline{\alpha_n}$ are fresh it is clear that $\mathcal{A}\pi'\pi_g = \mathcal{A}\pi'$ and $\alpha_i\pi'\pi_g = \alpha_i\pi_g = \tau_i$ for every type variable α_i. Then we can write the first derivation as $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi'\pi_g \vdash t : \tau'_t$ and by the Induction Hypothesis $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t$ and there is a type substitution π''_t such that $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi'\pi_g = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi''_t$ and $\tau_t\pi''_t = \tau'_t$. Since the data variables $\overline{X_n}$ do not appear in $\mathcal{A}\pi'$ then $\mathcal{A}\pi' = \mathcal{A}\pi'\pi_g = \mathcal{A}\pi_t\pi''_t$ and for every type variable $\alpha_i\pi'\pi_g = \alpha_i\pi_g = \tau_i = \alpha_i\pi_t\pi''_t$. Since $\mathcal{A}\pi' = \mathcal{A}\pi_t\pi''_t$ then we can write the second derivation as $\mathcal{A}\pi_t\pi''_t \vdash e_1 : \tau'_t$, and by the Induction Hypothesis $\mathcal{A}\pi_t \Vdash e_1 : \tau_1|\pi_1$ and there exists a type substitution π''_1 such that $\mathcal{A}\pi_t\pi''_t = \mathcal{A}\pi_t\pi_1\pi''_1$ and $\tau_1\pi''_1 = \tau'_t$. We can assume that π''_1 is minimal, so $Dom(\pi''_1) \subseteq FTV(\tau_1) \cup FTV(\mathcal{A}\pi_t\pi_1)$. Now we have to prove that $\tau_t\pi_1$ and τ_1 are unifiable, so there exists a most general unifier [63]. We define $B \equiv FTV(\pi''_t) \setminus FTV(\mathcal{A}\pi_t)$ and $\pi_u \equiv \pi''_1 + \pi''_t|_B$, which is well defined because the domains of the two components are disjoints. According to Observation 5, the variables of $FTV(\tau_1)$, $Dom(\pi_1)$ or $Rng(\pi_1)$ which do not occur in $FTV(\mathcal{A}\pi_t)$ will be fresh variables, so they will not be any of the variables in B. As the variables in B occur neither in $FTV(\mathcal{A}\pi_t)$ nor in $Rng(\pi_1)$, then they do not appear in $\mathcal{A}\pi_1\pi_1$; and as π''_1 is minimal then no variable in B occurs in $Dom(\pi''_1)$.

π_u is an unifier of $\tau_t\pi_1$ and τ_1 because $\tau_t\pi_1\pi_u = \tau_1\pi_u = \tau'_t$. The first case is easy because $\tau_1\pi_u = \tau_1\pi''_1 = \tau'_t$. To prove the second we will distinguish over the type variables of τ_t. For the type variables of τ_t in $\mathcal{A}\pi_t$ (i.e. those which are not in B) we know that $\tau_t\pi_1\pi_u = \tau_t\pi_1\pi''_1 = \tau_t\pi''_t = \tau'_t$, and for the others (those in B) we know they are fresh and do not appear in π_1, so $\tau_t\pi_1\pi_u = \tau_t\pi_u = \tau_t\pi''_t|_B = \tau'_t$. Therefore there will exist a most general unifier π, and $\pi_u = \pi\pi_o$.

We also know that $\mathcal{A}\pi' = \mathcal{A}\pi_t\pi''_t = \mathcal{A}\pi_t\pi_1\pi''_1 = \mathcal{A}\pi_t\pi_1\pi_u = \mathcal{A}\pi_t\pi_1\pi\pi_o$ and for every type variable $\alpha_i\pi_t\pi_1\pi\pi_o = \tau_i$ (for the type variables of $\alpha_i\pi_t$ which are in $\mathcal{A}\pi_t$ then $\alpha_i\pi_t\pi_1\pi\pi_o = \alpha_i\pi_t\pi_1\pi_u = \alpha_i\pi_t\pi_1\pi''_1 = \alpha_i\pi_t\pi''_t = \tau_i$, and for the rest of the variables -those in B- then $\alpha_i\pi_t\pi_1\pi\pi_o = \alpha_i\pi_t\pi_1\pi_u = \alpha_i\pi_t\pi_u = \alpha_i\pi_t\pi''_t|_B = \tau_i$).

Then we can write the third derivation as $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi\pi_o \vdash e_2 : \tau'$, and by the Induction Hypothesis $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2|\pi_2$ and there exists a type substitution π''_2 such that $\tau_2\pi''_2 = \tau'$ and $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi\pi_o = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi\pi_2\pi''_2$. Since the variables $\overline{X_n}$ do not appear in $\mathcal{A}$, in particular it is true that $\mathcal{A}\pi_t\pi_1\pi\pi_o = \mathcal{A}\pi_t\pi_1\pi\pi_2\pi''_2$.

With these three type inferences we can build the type inference for the let expression:

$$\textbf{[iLET}_m\textbf{]}\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t \\ \mathcal{A}\pi_t \Vdash e_1 : \tau_1|\pi_1 \\ (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2|\pi_2\end{array}}{\mathcal{A} \Vdash let_m\ t = e_1\ in\ e_2 : \tau_2|\pi_t\pi_1\pi\pi_2}$$

being $\pi = mgu(\tau_t\pi_1, \tau_1)$. To finish this case we only have to prove that there exists a type substitution π'' such that $\tau_2\pi'' = \tau'$ and $\mathcal{A}\pi' = \mathcal{A}\pi_t\pi_1\pi\pi_2\pi''$. This substitution π'' is π''_2.

[LET$^X_{pm}$] We assume that X does not occur in $\mathcal{A}$. We have a type derivation:

$$\textbf{[LET}^X_{pm}\textbf{]}\frac{\begin{array}{c}\mathcal{A}\pi' \vdash e_1 : \tau_1' \\ \mathcal{A}\pi' \oplus \{X : Gen(\tau_1', \mathcal{A}\pi')\} \vdash e_2 : \tau_2'\end{array}}{\mathcal{A}\pi' \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau_2'}$$

By the Induction Hypothesis we have that $\mathcal{A} \Vdash e_1 : \tau_1|\pi_1$ and there exists a type substitution π_1'' such that $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi_1''$ and $\tau_1\pi_1'' = \tau_1'$. $Gen(\tau_1', \mathcal{A}\pi') = Gen(\tau_1\pi_1'', \mathcal{A}\pi_1\pi_1'')$, so by Lemma 9 we know that $Gen(\tau_1, \mathcal{A}\pi_1)\pi_1'' \succ Gen(\tau_1', \mathcal{A}\pi')$. Then by Theorem 1-d the type derivation $\mathcal{A}\pi' \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\pi_1''\} \vdash e_2 : \tau_2'$ is valid. We can write this derivation as $(\mathcal{A}\pi_1 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\})\pi_1'' \vdash e_2 : \tau_2'$ and applying the Induction Hypothesis we obtain that $\mathcal{A}\pi_1 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\} \Vdash e_2 : \tau_2|\pi_2$ and there exists a type substitution π_2'' such that $\tau_2\pi_2'' = \tau_2'$ and $(\mathcal{A}\pi_1 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\})\pi_2\pi_2'' = (\mathcal{A}\pi_1 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\})\pi_1''$. Since X does not appear in $\mathcal{A}$ the last equality means that $\mathcal{A}\pi_1\pi_2\pi_2'' = \mathcal{A}\pi_1\pi_1''$ and $Gen(\tau_1, \mathcal{A}\pi_1)\pi_2\pi_2'' = Gen(\tau_1, \mathcal{A}\pi_1)\pi_1''$. With the previous type inferences we can construct a type inference for the whole expression:

$$\textbf{[iLET}^X_{pm}\textbf{]}\frac{\begin{array}{c}\mathcal{A} \Vdash e_1 : \tau_1|\pi_1 \\ \mathcal{A}\pi_1 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\} \Vdash e_2 : \tau_2|\pi_2\end{array}}{\mathcal{A} \Vdash let_{pm}\ X = e_1\ in\ e_2 : \tau_2|\pi_1\pi_2}$$

In this case it is easy to see that there exists a type substitution (π_2'') such that $\tau_2\pi_2'' = \tau_2'$ and $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi_1'' = \mathcal{A}\pi_1\pi_2\pi_2''$.

[LET$^h_{pm}$] Equal to the [LET$_m$] case.

[LET$_p$] The proof of this case follows the same ideas as the cases **[LET$_m$]** and **[LET$^X_{pm}$]**.

□

A result similar to the previous theorem is not possible with $\Vdash^\bullet$. The next theorem states that if the inference $\Vdash^\bullet$ succeeds then if finds a most general type and substitution (*b*). It also states that if this most general substitution exists, the inference succeeds (*a*) and vice versa.

Theorem 6 (Maximality of $\Vdash^\bullet$).

a) ${}^\bullet\Pi^e_{\mathcal{A}}$ has a maximum element $\Longleftrightarrow \exists \tau_g \in SType, \pi_g \in \mathcal{TS}ubst.\mathcal{A} \Vdash^\bullet e : \tau_g|\pi_g$.

b) If $\mathcal{A}\pi' \vdash^\bullet e : \tau'$ and $\mathcal{A} \Vdash^\bullet e : \tau|\pi$ then exists a type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ and $\tau' = \tau\pi''$.

Proof.
a)

$\Longleftarrow$) If $\mathcal{A} \Vdash^{\bullet} e : \tau_g|\pi_g$ then by Lemma 10 $\Pi^e_{\mathcal{A}} = {}^{\bullet}\Pi^e_{\mathcal{A}}$. Since $\mathcal{A} \Vdash e : \tau_g|\pi_g$ (by definition of $\Vdash^{\bullet}$) by Theorem 9 we know that $\Pi^e_{\mathcal{A}}$ has a maximum element, and also ${}^{\bullet}\Pi^e_{\mathcal{A}}$.

$\Longrightarrow$) We will prove that $\mathcal{A} \not\Vdash^{\bullet} e : \tau_g|\pi_g \Longrightarrow {}^{\bullet}\Pi^e_{\mathcal{A}}$ has not a maximum element.

(A) $\mathcal{A} \not\Vdash^{\bullet} e : \tau_g|\pi_g$ because $\mathcal{A} \not\Vdash e : \tau_g|\pi_g$. We know from Theorem 9 that if $\mathcal{A} \not\Vdash e : \tau_g|\pi_g$ then $\Pi^e_{\mathcal{A}}$ has not a maximum element. Then by Theorem 5 it cannot exists any type derivation $\mathcal{A}\pi' \vdash e : \tau'$, so $\Pi^e_{\mathcal{A}}$ is empty. Since ${}^{\bullet}\Pi^e_{\mathcal{A}} \subseteq \Pi^e_{\mathcal{A}}$ then ${}^{\bullet}\Pi^e_{\mathcal{A}} = \emptyset$ and cannot contain any maximum element.

(B) $\mathcal{A} \not\Vdash^{\bullet} e : \tau_g|\pi_g$ because $\mathcal{A} \Vdash e : \tau_g|\pi_g$ but $critVar_{\mathcal{A}\pi_g}(e) \neq \emptyset$. We will proceed by case distintion over the cause of the critical variables:

(B.1) $critVar_{\mathcal{A}\pi_g}(e) \neq \emptyset$ because for every pattern t_j in e and for every variable X_i in t_j that is critical then the cause of the opacity are type variables which appear in $\mathcal{A}\pi_g$. In other words, for those variables X_i then $\mathcal{A} \oplus \{X_m : \alpha_m\} \Vdash t_j : \tau_j|\pi_j$ and $FTV(\alpha_i\pi_j) \nsubseteq FTV(\tau_j)$ and $FTV(\alpha_i\tau_j) \setminus FTV(\tau_j) \subseteq FTV(\mathcal{A}\pi_g)$. It is clear that we can apply a type substitution to $\mathcal{A}\pi_g$ and eliminate the opacity of these variables. In particular we will always be able to find two type substitions π_1 and π_2 such that:

i. $\mathcal{A}\pi_g\pi_1 \vdash e : \tau_1$ and $\mathcal{A}\pi_g\pi_2 \vdash e : \tau_2$.
ii. $critVar_{\mathcal{A}\pi_g\pi_1}(e) = \emptyset$ and $critVar_{\mathcal{A}\pi_g\pi_2}(e) = \emptyset$
iii. No substitution π more general than $\pi_g\pi_1$ and $\pi_g\pi_2$ is in ${}^{\bullet}\Pi^e_{\mathcal{A}}$ because $critVar_{\mathcal{A}\pi}(e) = \emptyset$.

Let be $\overline{\beta_k}$ all the type variables causing opacity, and τ^1 and τ^2 two non unifiable types ($bool$ and $char$, for example). Then we can define $\pi_1 \equiv [\overline{\beta_k/\tau^1}]$ and $\pi_2 \equiv [\overline{\beta_k/\tau^2}]$. Since $\mathcal{A} \Vdash e : \tau_g|\pi_g$ by Theorem 4 $\mathcal{A}\pi_g \vdash e : \tau_g$, and by Theorem 1-a $\mathcal{A}\pi_g\pi_1 \vdash e : \tau_g\pi_1$ and $\mathcal{A}\pi_g\pi_2 \vdash e : \tau_g\pi_2$. We have eliminated the cause of opacity, so $critVar_{\mathcal{A}\pi_g\pi_1}(e) = \emptyset$ and $critVar_{\mathcal{A}\pi_g\pi_2}(e) = \emptyset$, i.e., $\pi_g\pi_1, \pi_g\pi_2 \in {}^{\bullet}\Pi^e_{\mathcal{A}}$. Finally since τ^1 and τ^2 are not unifiable, the only substitution more general that $\pi_g\pi_1$ and $\pi_g\pi_2$ that could be in ${}^{\bullet}\Pi^e_{\mathcal{A}}$ is π_g (substitutions more general than π_g cannot be in $\Pi^e_{\mathcal{A}}$, and neither in ${}^{\bullet}\Pi^e_{\mathcal{A}}$). But π_g is not in ${}^{\bullet}\Pi^e_{\mathcal{A}}$ because $critVar_{\mathcal{A}\pi_g}(e) \neq \emptyset$. Therefore ${}^{\bullet}\Pi^e_{\mathcal{A}}$ cannot have a maximum element because we have found two elements in ${}^{\bullet}\Pi^e_{\mathcal{A}}$ that do not have any "greater" element in ${}^{\bullet}\Pi^e_{\mathcal{A}}$.

(B.2) $critVar_{\mathcal{A}\pi_g}(e) \neq \emptyset$ because there exists some pattern t_j in e in which there is any variable X that is opaque because of type variables that do not occur in $\mathcal{A}\pi_g$. Intuitively

in this case these type variables will have appeared because of there exist a symbol in t_j whose type is a type-scheme, and that fresh variables come from the fresh variant used. From Theorem 5 we know that for every π_e in $\Pi^e_{\mathcal{A}}$ then $\mathcal{A}\pi_e = \mathcal{A}\pi_g\pi''$ for some type substitution π'. But $critVar_{\mathcal{A}\pi_e}(e) = critVar_{\mathcal{A}\pi_g\pi''}(e) \neq \emptyset$, because we always have fresh type variables causing opacity (since they come from type-schemes, substitutions do not affect them). Therefore for every $\pi_e \in \Pi^e_{\mathcal{A}}$ then $critVar_{\mathcal{A}\pi_e}(e) \neq \emptyset$, and as ${}^{\bullet}\Pi^e_{\mathcal{A}} \subseteq \Pi^e_{\mathcal{A}}$ then ${}^{\bullet}\Pi^e_{\mathcal{A}} = \emptyset$; so it has not a maximum element.

---

b) By definition of $\vdash^{\bullet}$ and $\Vdash^{\bullet}$ we know that $\mathcal{A}\pi' \vdash e : \tau'$ and $\mathcal{A} \Vdash e : \tau|\pi$. Then by Theorem 5 we know that exists a type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ and $\tau' = \tau\pi''$. □

The theorem that follows states that the substitution found by the procedure $\mathcal{B}$ for inferring types for programs make the program well-typed.

Theorem 7 (Soundness of $\mathcal{B}$).
$\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi \Longrightarrow wt_{\mathcal{A}\pi}(\mathcal{P})$.

Proof. Since $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ then we know that $\mathcal{A} \Vdash^{\bullet} (\varphi(r_1), \ldots, \varphi(r_m)) : (\tau_1, \ldots, \tau_m)|\pi$, and by Theorem 4 then $\mathcal{A}\pi \vdash^{\bullet} (\varphi(r_1), \ldots, \varphi(r_m)) : (\tau_1, \ldots, \tau_m)$. In order to prove $wt_{\mathcal{A}\pi}(\mathcal{P})$ we need to prove that every rule $r_i \equiv f_i\ t_1 \ldots t_n \rightarrow e_i$ in $\mathcal{P}$ is well-typed wrt. $\mathcal{A}\pi$. From Lemma 11 we know that $\mathcal{A}\pi \vdash^{\bullet} \varphi(r_i) : \tau_i$, so $\mathcal{A}\pi \vdash^{\bullet} pair\ \lambda t_1 \ldots t_n.e_i\ f_i : \tau_i$. This derivation can only be constructed if $\mathcal{A}\pi \vdash^{\bullet} \lambda t_1 \ldots t_n.e_i : \tau_i$ and $\mathcal{A}\pi \vdash^{\bullet} f_i : \tau_i$, and as the last derivation is just an application of rule **[ID]**, $\mathcal{A}\pi(f_i) \succ \tau_i$. We will distinguish between the case that $\mathcal{A}(f_i)$ is a simple type or a closed type-scheme:

a) If $\mathcal{A}(f_i)$ is a simple type, then $\mathcal{A}\pi(f_i)$ too. In this case $\mathcal{A}\pi(f_i) \succ \tau_i$ can only be true if $\mathcal{A}\pi(f_i) = \tau_i$, so trivially τ_i is a variant of $\mathcal{A}\pi(f_i)$. Therefore $\mathcal{A}\pi \vdash^{\bullet} \lambda t_1 \ldots t_n.e_i : \tau_i$ and τ_i is a variant of $\mathcal{A}\pi(f_i)$, so rule r_i is well-typed wrt. $\mathcal{A}\pi$.

b) $\mathcal{A}(f_i)$ is a closed type scheme, so $\mathcal{A}(f_i) = \mathcal{A}\pi(f_i)$. From step 2.- of $\mathcal{B}$ we know that in this case τ_i is a variant of $\mathcal{A}(f_i)$, and also of $\mathcal{A}\pi(f_i)$. Then since $\mathcal{A}\pi \vdash^{\bullet} \lambda t_1 \ldots t_n.e_i : \tau_i$ rule r_i is well-typed wrt. $\mathcal{A}\pi$.

□

The next theorem states that if the procedure $\mathcal{B}$ finds a substitution, it will be more general that any other substitution which makes the program well-typed.

Theorem 8 (Maximality of $\mathcal{B}$).
If $wt_{\mathcal{A}\pi'}(\mathcal{P})$ and $\mathcal{B}(\mathcal{A},\mathcal{P}) = \pi$ then $\exists\pi''$ such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$.

Proof. Since $wt_{\mathcal{A}\pi'}(\mathcal{P})$ we know that for every rule $r_i \equiv f_i\ t_1 \ldots t_n \rightarrow e_i$ in $\mathcal{P}$ there exists a type derivation $\mathcal{A}\pi' \vdash^\bullet \lambda t_1 \ldots t_n.e_i : \tau_i'$ and τ_i' is a variant of the type $\mathcal{A}\pi'(f_i)$. Then $\mathcal{A}\pi' \vdash^\bullet f_i : \tau_i'$, and we can construct type derivations $\mathcal{A}\pi' \vdash^\bullet pair\ \lambda t_1 \ldots t_n.e_i\ f_i : \tau_i'$. With these derivations we can build $\mathcal{A}\pi' \vdash^\bullet (\varphi(r_1),\ldots,\varphi(r_m)) : (\tau_1',\ldots,\tau_m')$ by Lemma 11. From $\mathcal{B}(\mathcal{A},\mathcal{P}) = \pi$ we know that $\mathcal{A} \Vdash^\bullet (\varphi(r_1),\ldots,\varphi(r_m)) : (\tau_1,\ldots,\tau_m)|\pi$, so by Theorem 6-*b* there will exist some type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$. □

This last theorem states that if there exists a most general substitution which gives type to e wrt. $\vdash$, then the inference $\Vdash$ will succeed, and vice versa.

Theorem 9 (Maximality of $\Vdash$)**.**
$\Pi^e_{\mathcal{A}}$ has a maximum element $\pi \iff \exists\tau_g \in SType, \pi_g \in \mathcal{TS}ubst.\mathcal{A} \Vdash e : \tau_g|\pi_g$.

Proof.

$\Longrightarrow$) If $\Pi^e_{\mathcal{A}}$ has maximum element π then there will be some type τ such that $\mathcal{A}\pi \vdash e : \tau$. Then by Theorem 5 we know that $\mathcal{A} \Vdash e : \tau_g|\pi_g$.

$\Longleftarrow$) We know from Theorem 5 that for every type substitution $\pi' \in \Pi^e_{\mathcal{A}}$ there exists a type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$. Then $\pi|_{FTV(\mathcal{A})} \lesssim \pi'$. From Theorem 4 we know that $\pi|_{FTV(\mathcal{A})}$ is in $\Pi^e_{\mathcal{A}}$, so it is the maximum element.

□